HelixCore

Helix Core C/C++ Developer Guide

2020.2

November 2020

PERFORCE

www.perforce.com

Contents

How to use this guide **13**

Syntax conventions 13

Feedback 14

Other documentation 14

Earlier versions of this guide 14

Overview **15**

Release compatibility of the API 15

Purpose of the API 15

Architecture of the API 15

Types of API files 15

Downloading the API archive 15

Compatibilities and version requirements 16

Compiling and linking Helix server applications **17**

Additional script libraries for Helix Core Extensions 17

Link order 17

SSL support 17

OpenSSL Library Version 17

Link order for SSL support 18

Compiler support 18

UNIX 18

Linux 18

Windows 19

Sample Jamfile 19

Sample Makefile 19

Application Programming **21**

Building with Jam 21

Building the sample application 22

Sending commands to the versioning service 24

Helix server settings on the user's machine 24

Connecting to the server 25

Displaying Helix server forms 25

Sending commands 26

Processing data from the server 26
Disconnecting from the server 29
Performing file I/O 29
Handling errors 33
Connection errors 34
Server errors 34
Find errorIds 34
Class overviews 35
ClientApi - Helix server connections and commands 35
ClientProgress - progress indicators for Helix server commands 36
ClientUser - I/O for Helix server commands 36
Error - collect and report layered errors 36
ErrorLog - output error messages 37
FileSys - Helix server file I/O 37
Ignore - support for rejecting files 37
KeepAlive - support for client-side disconnection 37
MapApi - logic for view mappings 38
Options - parse and store command line options 38
ServerHelperApi - helper methods for creating personal servers 38
Signaler - interrupt handling 38
StrBuf - string manipulation 38
StrDict - field/value manipulation 39
StrNum - small numeric strings 39
StrOps - string operations 39
StrPtr - text operations 39
StrRef - refer to existing strings 40
MonItems - between real-time counters and storage 40
MonItem - base class for performance counters 41
MonInteger - store 64-bit integer values 42
MonIntMax - tracks the highest recorded value 42
Public Methods Reference 44
ClientApi methods 49
ClientApi::DefineClient(const char *, Error *) 49
ClientApi::DefineHost(const char *, Error *) 50
ClientApi::DefineIgnoreFile(const char *, Error *) 51
ClientApi::DefinePassword(const char *, Error *) 51
ClientApi::DefinePort(const char *, Error *) 52

ClientApi::DefineUser(const char *, Error *) 53
ClientApi::Dropped() 54
ClientApi::Final(Error *) 55
ClientApi::GetClient() 55
ClientApi::GetConfig() 56
ClientApi::GetCwd() 57
ClientApi::GetHost() 58
ClientApi::GetIgnore() 59
ClientApi::GetIgnoreFile() 59
ClientApi::GetOs() 60
ClientApi::GetPassword() 61
ClientApi::GetPort() 62
ClientApi::GetProtocol(const char *) 63
ClientApi::GetTrans( ) 64
ClientApi::GetUser() 66
ClientApi::Init(Error *) 66
ClientApi::Run(const char *, ClientUser *) 67
ClientApi::SetBreak(KeepAlive *breakCallback) 68
ClientApi::SetClient(const StrPtr *) 70
ClientApi::SetClient(const char *) 71
ClientApi::SetCwd(const StrPtr *) 71
ClientApi::SetCwd(const char *) 72
ClientApi::SetCwdNoReload(const StrPtr *) 73
ClientApi::SetCwdNoReload(const char *) 74
ClientApi::SetHost(const StrPtr *) 75
ClientApi::SetHost(const char *) 75
ClientApi::SetIgnoreFile(const StrPtr *) 76
ClientApi::SetIgnoreFile(const char *) 77
ClientApi::SetPassword(const StrPtr *) 78
ClientApi::SetPassword(const char *) 79
ClientApi::SetPort(const StrPtr *) 79
ClientApi::SetPort(const char *) 80
ClientApi::SetProg(const StrPtr *) 80
ClientApi::SetProg(const char *) 81
ClientApi::SetProtocol(char *, char *) 82
ClientApi::SetProtocolV(char *) 85
ClientApi::SetTicketFile(const StrPtr *) 85

ClientApi::SetTicketFile(const char *) 86
ClientApi::SetTrans(int value) 87
ClientApi::SetTrustFile(const char *c) 87
ClientApi::SetTrustFile(const StrPtr *c) 88
ClientApi::SetUi(ClientUser *) 88
ClientApi::SetUser(const StrPtr *) 89
ClientApi::SetUser(const char *) 90
ClientApi::SetVersion(const StrPtr *) 91
ClientApi::SetVersion(const char *) 92
ClientProgress methods 93
ClientProgress::Description(const StrPtr *, int) 93
ClientProgress::Done(int) 94
ClientProgress::Total(long) 95
ClientProgress::Update(long) 96
ClientUser methods 96
ClientUser::CreateProgress(int) 97
ClientUser::Diff(FileSys *, FileSys *, int, char *, Error *) 97
ClientUser::Diff(FileSys *, FileSys *, FileSys *, int, char *, Error *) 100
ClientUser::Edit(FileSys *, Error *) 100
ClientUser::ErrorPause(char *, Error *) 101
ClientUser::File(FileSysType) 102
ClientUser::Finished() 103
ClientUser::HandleError(Error *) 104
ClientUser::HandleUrl( ) 104
ClientUser::Help(const char *const *) 105
ClientUser::InputData(StrBuf *, Error *) 106
ClientUser::Merge(FileSys *, FileSys *, FileSys *, FileSys *, Error *) 107
ClientUser::Message(Error *) 109
ClientUser::OutputBinary(const char *, int) 110
ClientUser::OutputError(const char *) 110
ClientUser::OutputInfo(char, const char *) 111
ClientUser::OutputStat(StrDict *) 113
ClientUser::OutputStatPartial( ) 115
ClientUser::OutputText(const char *, int) 115
ClientUser::ProgressIndicator() 116
ClientUser::Prompt(const StrPtr &, StrBuf &, int, Error *) 117
ClientUser::RunCmd(const char *, const char *, […], Error *) 118

Error methods 119
Error::Clear() 119
Error::Dump(const char *) 120
Error::Fmt(StrBuf *) 121
Error::Fmt(StrBuf *, int) 122
Error::GetGeneric() 123
Error::GetSeverity() 124
Error::IsFatal() 124
Error::IsWarning() 125
Error::Net(const char *, const char *) 126
Error::operator << (int) 126
Error::operator << (char *) 127
Error::operator << (const StrPtr &) 127
Error::operator = (Error &) 128
Error::Set(enum ErrorSeverity, const char *) 128
Error::Set(ErrorId &) 129
Error::Sys(const char *, const char *) 129
Error::Test() 130
ErrorLog methods 130
ErrorLog::Abort() 130
ErrorLog::Report() 131
ErrorLog::SetLog(const char *) 132
ErrorLog::SetSyslog() 132
ErrorLog::SetTag(const char *) 133
ErrorLog::UnsetSyslog() 134
FileSys methods 134
FileSys::Chmod(FilePerm, Error *) 134
FileSys::Close(Error *) 136
FileSys::Create(FileSysType) 137
FileSys::Open(FileOpenMode, Error *) 137
FileSys::Read(const char *, int, Error *) 139
FileSys::Rename(FileSys *, Error *) 140
FileSys::Set(const StrPtr *) 141
FileSys::Stat() 142
FileSys::StatModTime() 144
FileSys::Truncate() 145
FileSys::Unlink(Error *) 145

FileSys::Write(const char *, int, Error *) 146
Ignore methods 147
Ignore::Reject(const StrPtr &, const StrPtr &) 147
Ignore::RejectCheck(const StrPtr &) 148
KeepAlive methods 149
KeepAlive::IsAlive() 150
MapApi methods 151
MapApi::Clear() 151
MapApi::Count() 152
MapApi::GetLeft(int) 153
MapApi::GetRight(int) 153
MapApi::GetType(int) 154
MapApi::Insert(const StrPtr &, MapType) 155
MapApi::Insert(const StrPtr &, const StrPtr &, MapType) 155
MapApi::Join(MapApi *, MapApi *) [static] 156
MapApi::Join(MapApi *, MapDir, MapApi *, MapDir) [static] 158
MapApi::Translate(const StrPtr &, StrBuf&, MapDir) 159
Options methods 160
Options::GetValue(char opt, int subopt) 160
Options::operator[](char opt) 162
Options::Parse(int &, char ** &, const char *, int, const ErrorId &, Error *) 164
Options::Parse(int &, StrPtr * &, const char *, int, const ErrorId &, Error *) 171
ServerHelperApi methods 171
ServerHelperApi::ClearProtocol() 172
ServerHelperApi::CloneFromRemote(int,const StrPtr*,ClientUser*,Error*) 172
ServerHelperApi::CloneFromRemote(int,const char*,ClientUser*,Error*) 174
ServerHelperApi::CopyConfiguration(ServerHelperApi*,ClientUser*,Error*) 174
ServerHelperApi::Exists(ClientUser*,Error*) 175
ServerHelperApi::GetCaseFlag() 176
ServerHelperApi::GetClient() 176
ServerHelperApi::GetClient(Error*) 176
ServerHelperApi::GetDvcsDir() 177
ServerHelperApi::GetPort() 178
ServerHelperApi::GetProg() 178
ServerHelperApi::GetQuiet() 179
ServerHelperApi::GetUnicode() 179
ServerHelperApi::GetUser() 179

ServerHelperApi::GetVersion() 180
ServerHelperApi::InitLocalServer(ClientUser*,Error*) 180
ServerHelperApi::PrepareToCloneFilepath(ServerHelperApi*,const char*,ClientUser*,Error*) 182
ServerHelperApi::PrepareToCloneFilepath(ServerHelperApi*,const StrPtr*,ClientUser*,Error*) 183
ServerHelperApi::PrepareToCloneRemote(ServerHelperApi*,const char*,ClientUser*,Error*) 183
ServerHelperApi::PrepareToCloneRemote(ServerHelperApi*,const StrPtr*,ClientUser*,Error*) 184
ServerHelperApi::SetCaseFlag(const char*,Error*) 185
ServerHelperApi::SetCaseFlag(const StrPtr*,Error*) 185
ServerHelperApi::SetClient(const char*) 186
ServerHelperApi::SetClient(const StrPtr*) 186
ServerHelperApi::SetDefaultStream(const char*) 186
ServerHelperApi::SetDefaultStream(const StrPtr*) 187
ServerHelperApi::SetDvcsDir(const char*) 187
ServerHelperApi::SetDvcsDir(const StrPtr*) 188
ServerHelperApi::SetPort(const char*) 188
ServerHelperApi::SetPort(const StrPtr*) 188
ServerHelperApi::SetProg(const char*) 189
ServerHelperApi::SetProg(const StrPtr*) 189
ServerHelperApi::SetProtocol(const char*,const char*) 190
ServerHelperApi::SetProtocolV(const char*) 190
ServerHelperApi::SetQuiet() 190
ServerHelperApi::SetUnicode(int) 191
ServerHelperApi::SetUser(const char*) 191
ServerHelperApi::SetUser(const StrPtr*) 191
ServerHelperApi::SetVersion(const char*) 192
ServerHelperApi::SetVersion(const StrPtr*) 192
Signaler methods 193
Signaler::Block() 193
Signaler::Catch() 194
Signaler::DeleteOnIntr(void *) 195
Signaler::Intr() 197
Signaler::OnIntr(SignalFunc, void *) 200
Signaler::Signaler() (constructor) 201
StrBuf methods 202

StrBuf::Alloc(int) 202
StrBuf::Append(const char *) 204
StrBuf::Append(const char *, int) 205
StrBuf::Append(const StrPtr *) 207
StrBuf::Clear() 209
StrBuf::StrBuf() (Constructor) 210
StrBuf::StrBuf(const StrBuf &) (Copy Constructor) 211
StrBuf::~StrBuf() (Destructor) 212
StrBuf::Extend(char) 213
StrBuf::Extend(const char *, int) 215
StrBuf::operator =(const char *) 216
StrBuf::operator =(const StrBuf &) 217
StrBuf::operator =(const StrPtr &) 218
StrBuf::operator =(const StrRef &) 219
StrBuf::operator <<(const char *) 220
StrBuf::operator <<(int) 221
StrBuf::operator <<(const StrPtr *) 223
StrBuf::operator <<(const StrPtr &) 224
StrBuf::Set(const char *) 226
StrBuf::Set(const char *, int) 227
StrBuf::Set(const StrPtr *) 228
StrBuf::Set(const StrPtr &) 229
StrBuf::StringInit() 230
StrBuf::Terminate() 233
StrDict methods 235
StrDict::GetVar(const StrPtr &) 235
StrDict::GetVar(const char *) 237
StrDict::GetVar(const char *, Error *) 237
StrDict::GetVar(const StrPtr &, int) 238
StrDict::GetVar(const StrPtr &, int, int) 238
StrDict::GetVar(int, StrPtr &, StrPtr &) 239
StrDict::Load(FILE *) 239
StrDict::Save(FILE *) 241
StrDict::SetArgv(int, char *const *) 241
StrNum methods 243
StrNum::StrNum(int) (constructor) 243
StrNum::Set(int) 243

StrOps methods 244
StrOps::Caps(StrBuf &) 244
StrOps::Dump(const StrPtr &) 245
StrOps::Expand(StrBuf &, StrPtr &, StrDict &) 246
StrOps::Expand2(StrBuf &, StrPtr &, StrDict &) 247
StrOps::Indent(StrBuf &, const StrPtr &) 249
StrOps::Lines(StrBuf &, char *[], int) 250
StrOps::Lower(StrBuf &) 251
StrOps::OtoX(const unsigned char *, int, StrBuf &) 252
StrOps::Replace(StrBuf &, const StrPtr &, const StrPtr &, const StrPtr &) 253
StrOps::Sub(StrPtr &, char, char) 254
StrOps::Upper(StrBuf &) 255
StrOps::Words(StrBuf &, const char *[], char *[], int) 256
StrOps::XtoO(char *, unsigned char *, int) 257
StrPtr methods 258
StrPtr::Atoi() 258
StrPtr::CCompare(const StrPtr &) 259
StrPtr::Compare(const StrPtr &) 260
StrPtr::Contains(const StrPtr &) 261
StrPtr::Length() 262
StrPtr::operator [](int) 263
StrPtr::operators ==, !=, >, <, <=, >= (const char *) 264
StrPtr::operators ==, !=, >, <, <=, >= (const StrPtr &) 265
StrPtr::Text() 266
StrPtr::Value() 267
StrPtr::XCompare(const StrPtr &) 267
StrRef methods 269
StrRef::StrRef() (constructor) 269
StrRef::StrRef(const StrPtr &) (constructor) 269
StrRef::StrRef(const char *) (constructor) 270
StrRef::StrRef(const char * , int) (constructor) 271
StrRef::Null() 272
StrRef::operator =(StrPtr &) 273
StrRef::operator =(char *) 274
StrRef::operator +=(int) 275
StrRef::Set(char *) 276
StrRef::Set(char * , int) 277

StrRef::Set(const StrPtr *) 278
StrRef::Set(const StrPtr &) 279
Glossary **281**
License Statements **300**

How to use this guide

This guide contains details about using the Helix C/C++ API to create applications that interact with Helix server. Be sure to also read the code in the API's header and C files. Interfaces for Java, Perl, Ruby, Python, PHP, and other languages are available from our website at https://www.perforce.com/downloads.

This section provides information on typographical conventions, feedback options, and additional documentation.

Syntax conventions

Helix documentation uses the following syntax conventions to describe command line syntax.

Notation	Meaning
`literal`	Must be used in the command exactly as shown.
italics	A parameter for which you must supply specific information. For example, for a *serverid* parameter, supply the ID of the server.
`-a -b`	Both *a* and *b* are required.
`{-a | -b}`	Either *a* or *b* is required. Omit the curly braces when you compose the command.
`[-a -b]`	Any combination of the enclosed elements is optional. None is also optional. Omit the brackets when you compose the command.
`[-a | -b]`	Any one of the enclosed elements is optional. None is also optional. Omit the brackets when you compose the command.
`...`	Previous argument can be repeated. ▪ `p4 [g-opts] streamlog [ -l -L -t -m max ] stream1 ...` means `1` or more stream arguments separated by a space ▪ See also the use on `...` in Command alias syntax in the *Helix Core P4 Command Reference* **Tip** `...` has a different meaning for directories. See Wildcards in the *Helix Core P4 Command Reference*.

Feedback

How can we improve this manual? Email us at manual@perforce.com.

Other documentation

See https://www.perforce.com/support/self-service-resources/documentation.

> **Tip**
> You can also search for Support articles in the Perforce Knowledgebase.

Earlier versions of this guide

2020.2, 2020.1, 2019.2, 2019.1, 2018.2, 2018.1, 2017.2, 2017.1, 2016.2, 2016.1, 2015.2, 2015.1, 2014.2, 2014.1

Overview

This section provides information on release compatibility, purpose, and architecture of the API.

Release compatibility of the API

The Helix C/C++ API is subject to change from release to release, and is not guaranteed to be source-code compatible from one release to the next. However, applications that you create using the API can run against previous releases of Helix server and should also run against later releases of Helix server.

Support for specific features depends on the version of Helix server and the API that you use.

Purpose of the API

The Helix C/C++ API enables you to create **applications** that:

- interact with end users
- send commands to the Helix server
- process the data the Helix server returns

Architecture of the API

The client session is managed by a C++ class called `ClientApi`. (See "ClientApi methods" on page 49)

User interaction is channeled through the `ClientUser` C++ class. The default methods of `ClientUser` implement the `p4` command line interface. To create a custom client application, create a subclasses of `ClientUser`. (See "ClientUser methods" on page 96)

Types of API files

The Helix C/C++ API consists of:

- header files (`.h`), which are in the `include/p4` subdirectory
- platform-specific link libraries in the `lib` subdirectory
- reference implementation of the `ClientUser` class, `clientuser.cc`, which is in the `sample` subdirectory

Downloading the API archive

The API is packaged as an .tgz or .zip file. To download for your platform, do one of the following:

- go to the Helix Core FTP site at ftp://ftp.perforce.com/perforce/. For example, at ftp://ftp.perforce.com/perforce/r19.1/bin.macosx105x86_64/, you can download the `p4api.tgz` file, which is the 2019.1 release of the API for the 10.5 version of MacOS
- go to https://www.perforce.com/downloads/helix-core-c/c-api

Compatibilities and version requirements

The P4API source code is compatible with the C++98 language standard, except for the optional Helix Core Extensions (see Helix Core Extensions Developer Guide), which requires C++17.

The pre-built API distributions require either `OpenSSL 1.0` or `OpenSSL 1.1`. The version of OpenSSL the API requires is in its file name. For example, `p4api_vs2013_dyn_openssl1.1.1.zip` requires the `1.1` version.

Similarly, versions must match for Windows VisualStudio.

For Linux platforms, the GLIBC version compatibility is part of the file name. For example, `p4api-glibc2.12-openssl1.1.1.tgz` was built with `1.1`

For Extensions-compatible builds, the `libstdc++` version must be at least `GLIBCXX_3.4.22`

For non-Windows platforms, the `Pthreads` library must be available.

Compiling and linking Helix server applications

The following sections tell you how to build your application on the target platform.

To build `p4api.cc`, include `clientapi.h`, which includes all the necessary header files for the sample client application.

Additional script libraries for Helix Core Extensions

To learn about this technology, see the *Helix Core Extensions Developer Guide*.

Name	Description
`libp4script.a`	For the code common to server and client
`libp4script_c.a`	For the client-side
`libp4script_curl`	If you already have a copy of curl, you can skip linking to this version
`libp4script_sqlite`	If you already have a copy of SQLite, you can skip linking to this version

Link order

The link libraries distributed with P4API must be linked explicitly in the following order:

	UNIX	Windows
1	`libclient.a`	`libclient.lib`
2	`librpc.a`	`librpc.lib`
3	`libsupp.a`	`libsupp.lib`

SSL support

P4API libraries must be linked against the OpenSSL libraries from http://www.openssl.org/.

OpenSSL Library Version

We recommend keeping current with the latest minor version matching the version referenced in the Helix C/C++ API file `librpc.a` (or `librpc.lib` on Windows). To see which version is referenced by the library, run the following command on UNIX variants or Macintosh:

```
strings librpc.a | grep ^OpenSSL
```

On Windows:

```
strings librpc.lib | findstr /B OpenSSL
```

This command will produce an output similar to the following:

OpenSSL 1.0.1p 9 Jul 2015

In this example, you would use the latest minor version of OpenSSL that matches version 1.0.1.

Link order for SSL support

To enable SSL support, link with the ssl and crypto libraries from OpenSSL. This results in the following link order:

- `libclient.a`
- `librpc.a`
- `libsupp.a`
- `libssl.a`
- `libcrypto.a`

On Windows, the ssl and crypto OpenSSL libraries are named `ssleay32.lib` and `libeay32.lib` respectively.

Compiler support

UNIX

For all UNIX platforms, you can use the `gcc` compiler to compile client applications with the Helix C/C++ API.

Note that `clientapi.h` includes `stdhdrs.h`, which might attempt to set platform-specific defines. To ensure these defines are set properly, compile with the `-DOS_XXX` flag, where *XXX* is the platform name as specified by Perforce. (Use `p4 -V` to display the platform name; for example, for `LINUX52X86`, specify -DOS_LINUX.)

Some platforms require extra link libraries for sockets. Solaris requires the following compiler flags:

-lsocket -lnsl

Linux

Some platforms require extra link libraries for runtime support. Linux requires the following compiler flag:

```
-lrt
```

Windows

Using Microsoft Visual Studio (VC++), compile your client application with the following flags:

`/DOS_NT /MT /DCASE_INSENSITIVE`

For debugging, compile with the **`/MTd`** flag for multithreading.
Do not compile with **`/MD`** or **`/MDd`**, because these flags can cause undefined behavior.

Link with the following libraries:

- **`libcmt.lib`**
- **`oldnames.lib`**
- **`kernel32.lib`**
- **`ws2_32.lib`**
- **`advapi32.lib`**

Sample Jamfile

The following example shows a Jamfile that can be used to build **`p4api.cc`**, a Helix server application. (This example is in the **`api`** subdirectory.)

```
CFLAGS = -g -D_GNU_SOURCE ;
LINK = c ;OPTIM = ;
Main p4api : p4api.cc ;
ObjectHdrs p4api : api ;
LinkLibraries p4api : api/libclient.a api/librpc.a api/libsupp.a
```

For more about **`jam`**, see "Building with Jam" on page 21.

Sample Makefile

The following is a GNU **`make`** file for building **`p4api.cc`**, a Helix server application. (The example assumes the API is installed in the **`api`** subdirectory.)

```
SOURCES = p4api.cc
INCLUDES = -Iapi
OBJECTS = ${SOURCES:.cc=.o}
LIBRARIES = api/libclient.a api/librpc.a api/libsupp.a
BINARY = p4api
RM = /bin/rm -f


C = c
```

```
CFLAGS = -c -g -D_GNU_SOURCE
LINK = c
LINKFLAGS =

.cc.o :
   ${C} ${CFLAGS} $< ${INCLUDES}

${BINARY} : ${OBJECTS}
   ${LINK} -o ${BINARY} ${OBJECTS} ${LIBRARIES}

clean :
   - ${RM} ${OBJECTS} ${BINARY}
```

Application Programming

This section provides information on application programming.

Building with Jam **21**
Building the sample application 22
Sending commands to the versioning service **24**
Helix server settings on the user's machine 24
Connecting to the server 25
Displaying Helix server forms 25
Sending commands 26
Processing data from the server 26
Disconnecting from the server 29
Performing file I/O **29**
Handling errors **33**
Connection errors 34
Server errors 34
Find errorIds 34
Class overviews **35**
ClientApi - Helix server connections and commands 35
ClientProgress - progress indicators for Helix server commands 36
ClientUser - I/O for Helix server commands 36
Error - collect and report layered errors 36
ErrorLog - output error messages 37
FileSys - Helix server file I/O 37
Ignore - support for rejecting files 37
KeepAlive - support for client-side disconnection 37
MapApi - logic for view mappings 38
Options - parse and store command line options 38
ServerHelperApi - helper methods for creating personal servers 38
Signaler - interrupt handling 38
StrBuf - string manipulation 38
StrDict - field/value manipulation 39
StrNum - small numeric strings 39
StrOps - string operations 39
StrPtr - text operations 39
StrRef - refer to existing strings 40
MonItems - between real-time counters and storage 40
MonItem - base class for performance counters 41
MonInteger - store 64-bit integer values 42
MonIntMax - tracks the highest recorded value 42

Building with Jam

Jam is a build tool, similar in its role to the more familiar `make`. Jamfiles are to `jam` as makefiles are to `make`.

Jam is an Open Source project sponsored by Perforce Software, Inc.. Jam documentation, source code, and links to precompiled binaries are available from the Jam product information page at:

http://www.perforce.com/documentation/jam

The P4API distribution contains the necessary header files (`*.h`) and libraries (`libclient.a`, `librpc.a`, `libsupp.a`, `libp4sslstub.a`) required to compile and link a client application. The distribution also includes a sample application in C++, `p4api.cc`.

In general, the process is similar to most APIs: compile your application sources, then link them with the API libraries. The precise steps needed vary somewhat from platform to platform.

The sample application `p4api.cc` is a portable, minimal Helix server application, which we can use as an example. For purposes of this example, assume a Linux system.

Compile and link `p4api.cc` as follows:

```
$ cc -c -o p4api.o -D_GNU_SOURCE -O2 -DOS_LINUX -DOS_LINUX24 \
> -DOS_LINUXX86 -DOS_LINUX24X86 -I. -Imsgs -Isupport -Isys p4api.cc

$ gcc -o p4api p4api.o libclient.a librpc.a libsupp.a libp4sslstub.a
```

The preprocessor definitions (-D*definition*) vary from platform to platform.

In order to build the example across a wide variety of platforms, the API distribution also contains two "Jamfiles" (`Jamrules` and `Jamfile`) that describe to how to build the sample application on each platform.

Building the sample application

Once you have Jam on your system, you can use it to build the `p4api` application. On some platforms, `jam` needs an extra hint about the operating system version. For instance, on RedHat Linux 7.1, with a 2.4 linux kernel, use `OSVER=24`:

```
$ jam
Set OSVER to 42/52 [RedHat M.n], or 22/24 [uname -r M.n]

$ uname -r
2.4.2-2

$ jam -s OSVER=24
...found 121 target(s)...
...updating 2 target(s)...
C++ p4api.o
Link p4api
Chmod1 p4api
```

```
...updated 2 target(s)...

$ p4api info
User name: you
Client name: you:home:sunflower
Client host: sunflower
Client root: /home/you
Current directory: /home/you/tmp/p4api
Client address: 207.46.230.220:35012
Server address: sunflower:1674
Server root: /home/p4/root
Server date: 2009/09/24 12:15:39 PDT
Server version: P4D/LINUX22X86/2009.1/192489 (2009/04/12)
Server license: Your Company 10 users (expires 2010/02/10)
Server license-ip: 10.0.0.2
```

As shown in the example above, **`jam`** does not, by default, show the actual commands used in the build (unless one of them fails). To see the exact commands **`jam`** generates, use the -o *file* option. This causes **`jam`** to write the updating actions to *file*, suitable for execution by a shell.

To illustrate; first, invoke **`jam clean`** to undo the build:

```
$ jam -s OSVER=42 clean
  ...found 1 target(s)...
  ...updating 1 target(s)...
  Clean clean
  ...updated 1 target(s)...
```

Then use **`jam -o build_sample`** to create the build file:

```
$ jam -s OSVER=42 -o build_sample
  ...found 121 target(s)...
  ...updating 2 target(s)...
  C++ p4api.o
  Link p4api
  Chmod1 p4api
  ...updated 2 target(s)...

$ cat build_sample
```

```
cc -c -o p4api.o -O2 -DOS_LINUX -DOS_LINUX42 -DOS_LINUXX86 \
-DOS_LINUX42X86 -I. -Imsgs -Isupport -Isys p4api.cc
gcc -o p4api p4api.o libclient.a librpc.a libsupp.a libp4sslstub.a
chmod 711 p4api
```

The generated **`build_sample`** can then be executed by a shell:

`/bin/sh build_sample`

to produce the executable, which you can test by running **`p4api info`** or most other Helix server commands:

```
$ p4api changes -m 1


Change 372 on 2002/09/23 by you@you:home:sunflower 'Building API'
```

As you can see, **`p4api`** is a usable full-featured command line Helix server client (very similar to the **`p4`** command). The example's functionality comes from the default implementation of the **`ClientUser`** class, linked from the **`libclient.a`** library and the rest of the library code, for which source code is not included. The source for the default implementation is provided in the P4API distribution as **`clientuser.cc`**.

Sending commands to the versioning service

Helix server applications interact with the versioning service by:

1. Initializing a connection.
2. Sending commands.
3. Closing the connection.

The Helix server does not maintain any kind of session identifier. The server identifies the sender of commands by its combination of Helix server user name and client workspace. Different processes that use the same combination of user and workspace are not distinguished by the Helix server. To prevent processes from interfering with each other when submitting changelists, be sure to use separate client specifications for each process. If you need to create large numbers of processes, consider creating a cache of client specifications and serving them to processes as required.

Helix server settings on the user's machine

To determine which server and depot are accessed and how files are mapped, the standard classes in the API observe the Helix server settings on the user's machine. Assuming the workstation is configured correctly, your application does not need to provide logic that specifies server, port, workspace, or user.

To override the user's settings, your application can call **`Set`** methods.

Settings take precedence as follows, highest to lowest:

1. Values set within a Helix server application
2. Values in configuration files (`P4CONFIG`)
3. Values set as environment variables at the operating system prompt
4. Variables residing in the registry (set using the `p4 set` or `p4 set -s` commands on Windows client machines)
5. Default values defined by Helix server or gathered from the system

Connecting to the server

To connect to the Helix server for which the client computer is configured, your client application must call the `client.Init()` method; for example:

```
client.Init( &e );
if ( e.Test() )
   {
      printf("Failed to connect:\n" );
      ErrorLog::Abort(); // Displays the error and exits
   }
printf( "Connected OK\n" );
```

Your program only needs to connect once. After connecting, the application can issue as many Helix server commands as required. If you intend to use tagged output, your program must call `client.SetProtocol()` before calling `client.Init()`. For details about using tagged output, refer to "Tagged data" on the next page.

Displaying Helix server forms

Helix server client commands that collect a large amount of input from the user (such as `p4 branch`, `p4 change`, `p4 label`) use ASCII forms. To interact with your end user, your client application program can display Helix server ASCII forms such as changelists, client specification, and so on. To display a form and collect user input, call `ClientUser::Edit()`, which puts the form into a temporary file and invokes the text editor that is configured for the client machine.

All form-related commands accept the batch mode flags -o and -i:

- -o causes the form to be passed to `ClientUser::OutputInfo()`.
- -i causes the form to be read with `ClientUser::InputData()`.

These flags allow changes to the form to occur between separate invocations of the `p4` application, rather than during a single invocation. (For details about the -o and -i global options, see the *P4 Command Reference*.)

All form-related commands can return a form descriptor. Your application can use this descriptor to parse forms into constituent variables and to format them from their constituent variables. The `specstring` protocol variable enables this support in the server. Form descriptors are best used with the `tag` protocol variable, which causes the form data to appear using `ClientUser::OutputStat()` rather than `OutputInfo()`.

Select the protocol with `ClientApi::SetProtocol()` as follows:

```
client.SetProtocol( "specstring", "" );
client.SetProtocol( "tag", "" );
```

To obtain the descriptor containing the results of the method call, your application must pass a `StrDict` object to `ClientUser::OutputStat()`. Your application can override the `OutputStat()` method in a class derived from `ClientUser`. The Helix C/C++ API calls this derived method, passing it the output from the command.

Sending commands

The following example illustrates how you set up arguments and execute the `p4 fstat` command on a file named `Jam.html`.

```
char file[] = "Jam.html";
char *filep = &file[0];
client.SetArgv( 1, &filep );
client.Run( "fstat", &ui );
```

For commands with more arguments, use an approach like the following:

```
char    *argv[] = { "-C", "-l", 0, 0 };
int     argc    = 2;
char    *file   = "Jam.html";
argv[ argc++ ]  = file;
client.SetArgv( argc, argv );
client.Run( "fstat", &ui );
```

Processing data from the server

Helix server can return tagged data (name-value pairs) for some commands. The following sections tell you how to handle tagged and untagged data.

Tagged data

The following example shows data returned in tagged format by `p4 -Ztag clients` command. (The -Z flag specifies that tagged data is to be returned; this flag is unsupported and intended for debugging use.)

```
...client xyzzy
...Update 972354556
...Access 970066832
...Owner gerry
...Host xyzzy
...Description Created by gerry
```

To enable the Helix server to return tagged data, your application must call `SetProtocol("tag", "")` before connecting to the server. To extract values from tagged data, use the `GetVars` method.

List of commands that can output tagged data

The following Helix server commands can return tagged output.

> **Note**
> The tagged output of some commands can changed across releases. For complete details, see the release notes for your version, such as: https://www.perforce.com/perforce/r18.1/user/p4devnotes.txt where `r18.1` represents the 2018.1 release.
>
> To obtain output in the form used by earlier revisions of Helix server, set the api variable according to the notes for `SetProtocol()`.

p4 add

p4 attribute

p4 branch -o

p4 branches

p4 change -o

p4 changes

p4 client -o

p4 clients

p4 counter

p4 counters

p4 delete

p4 depots

p4 describe

p4 diff - (summary line)

p4 diff2

p4 edit

p4 filelog

p4 files

p4 fix

p4 fixes

p4 group -o

p4 groups

p4 have

p4 info - can indicate `enabled` for the fields `password`, `monitor`, `security`, and `unicode`

p4 integrate

p4 integrated

p4 interchanges

p4 job -o

p4 jobs

p4 jobspec -o

p4 label -o

p4 labels

p4 labelsync

p4 lock

p4 logger

p4 monitor

p4 obliterate

p4 print

p4 protect -o

p4 reopen

p4 resolve -at, p4 resolve -ay, p4 resolve -n

p4 revert

p4 review

p4 reviews

p4 submit

p4 sync

p4 triggers -o

p4 typemap -o

p4 unlock

p4 user -o

p4 users - can indicate that the `password` field is set to enabled

p4 verify

p4 where

Untagged Data

To handle untagged data, create a subclass of `ClientUser` for every type of data required and provide alternate implementations of `ClientUser::OutputInfo()`, `OutputBinary()`, `OutputText()`, and `OutputStat()`.

Disconnecting from the server

After your application has finished interacting with the Helix server, it must disconnect as illustrated below:

```
client.Final( &e );
e.Abort();
```

To ensure the application can exit successfully, make sure to call `ClientApi::Final()` before calling the destructor.

Performing file I/O

The default client file I/O implementation returns a `FileSys` object, which is described in `filesys.h`. To intercept client workspace file I/O, replace the `FileSys *ClientUser::File()` method by subclassing `ClientUser`.

The following example illustrates how you can override `FileSys`.

```
#include "p4/clientapi.h"
class MyFileSys : public FileSys {
    public:

    MyFileSys();
    ~MyFileSys();

    virtual void    Open( FileOpenMode mode, Error *e );
    virtual void    Write( const char *buf, int len, Error *e );
    virtual int     Read( char *buf, int len, Error *e );
    virtual int     ReadLine( StrBuf *buf, Error *e );
    virtual void    Close( Error *e );
```

```
    virtual int     Stat();
    virtual int     StatModTime();
    virtual void    Truncate( Error *e );
    virtual void    Unlink( Error *e = 0 );
    virtual void    Rename( FileSys *target, Error *e );
    virtual void    Chmod( FilePerm perms, Error *e );

    protected:
    int nchars;
} ;

MyFileSys::MyFileSys()
{
    nchars = 0;
}

MyFileSys::~MyFileSys()
{
    printf( "Number of characters transferred = %d\n", nchars );
}

void MyFileSys::Open( FileOpenMode mode, Error *e )
{
    printf( "In MyFileSys::Open()\n" );
}

void MyFileSys::Write( const char *buf, int len, Error *e )
{
    printf( "In MyFileSys::Write()\n" );
    printf( "%s", buf );
    nchars = nchars + len;
}

int MyFileSys::Read( char *buf, int len, Error *e )
{
```

```
    printf( "In MyFileSys::Read()\n" );
    return 0;
}

int MyFileSys::ReadLine( StrBuf *buf, Error *e )
{
    printf( "In MyFileSys::ReadLine()\n" );
    return 0;
}
```

```
void MyFileSys::Close( Error *e )
{
    printf( "In MyFileSys::Close()\n" );
}

int MyFileSys::Stat()
{
    printf( "In MyFileSys::Stat()\n" );
    return 0;
}

int MyFileSys::StatModTime()
{
    printf( "In MyFileSys::StatModTime()\n" );
    return 0;
}

void MyFileSys::Truncate( Error *e )
{
    printf( "In MyFileSys::Truncate()\n" );
}

void MyFileSys::Unlink( Error *e = 0 )
{
    printf( "In MyFileSys::Unlink()\n" );
}
```

```
void MyFileSys::Rename( FileSys *target, Error *e )
{
    printf( "In MyFileSys::Rename()\n" );
}

void MyFileSys::Chmod( FilePerm perms, Error *e )
{
    printf( "In MyFileSys::Chmod()\n" );
}

class ClientUserSubclass : public ClientUser {
    public:
    virtual FileSys *File( FileSysType type );
};

FileSys *ClientUserSubclass::File( FileSysType type )
{
    return new MyFileSys;
}
```

```
int main( int argc, char **argv )
{
    ClientUserSubclass ui;
    ClientApi client;
    Error e;

    char force[] = "-f";
    char file[] = "hello.c";
    char *args[2] = { &force[0], &file[0] };

    // Connect to server

    client.Init( &e );
    e.Abort();
```

```
    // Run the command "sync -f hello.c"

    client.SetArgv( 2, &args[0] );
    client.Run( "sync", &ui );

    // Close connection

    client.Final( &e );
    e.Abort();
    return 0;
}
```

The preceding program produces the following output when you run it.

```
% ls -l hello.c
-r--r--r--    1 member   team           41 Jul 30 16:57 hello.c
% cat hello.c
main()
{
  printf( "Hello World!\n" );
}
% samplefilesys
//depot/main/hello.c#1 - refreshing /work/main/hello.c
In MyFileSys::Stat()
In MyFileSys::Open()
In MyFileSys::Write()
main()
{
   printf( "Hello World!\n" );
}
In MyFileSys::Close()
Number of characters transferred = 41
```

Handling errors

To encapsulate error handling in a maintainable way, subclass `ClientUser` at least once for every command you want to run and handle errors in the `HandleError()` method of the derived class.

To best handle the formatting of error text, parse the error text, looking for substrings of anticipated errors, and display the rest. For example:

```
void P4CmdFstat::HandleError(Error *e)
{
    StrBuf  m;
    e->Fmt( &m );
    if ( strstr( m.Text(), "file(s) not in client view." ) )
        e->Clear();
    else if ( strstr( m.Text(), "no such file(s)" ) )
        e->Clear();
    else if ( strstr( m.Text(), "access denied" ) )
        e->Clear();
    else
        this->e = *e;
}
```

Connection errors

If any error occurs when attempting to connect with the Helix server, the `ClientApi::Init()` method returns an error code in its `Error` parameter.

Server errors

The `ClientApi::Final()` method returns any I/O errors that occurred during `ClientApi::Run()` in its `Error` parameter. `ClientApi::Final()` returns a non-zero value if any I/O errors occurred or if `ClientUser::OutputError()` was called (reporting server errors) during the command run.

To report errors generated by the server during an operation, your application can call the `ClientUser::HandleError()` method. The default implementation of `HandleError()` is to format the error message and call `ClientUser::OutputError()`, which, by default, writes the message to standard output. `HandleError()` has access to the raw `Error` object, which can be examined with the methods defined in `error.h`. Prior to release 99.1, Helix server invoked `OutputError()` directly with formatted error text.

Find errorIds

You can find errorIds by running commands on the command line with the `-e` global option. This option displays more detailed information about each error or info message received, including the formatting strings, unique error codes, and severity values that are available to API users.

For more information on global options, see the *Helix Core P4 Command Reference*.

Class overviews

The following classes comprise the Helix C/C++ API. Public methods for these classes are documented in "Public Methods Reference" on page 44.

ClientApi - Helix server connections and commands 35
ClientProgress - progress indicators for Helix server commands 36
ClientUser - I/O for Helix server commands 36
Error - collect and report layered errors 36
ErrorLog - output error messages 37
FileSys - Helix server file I/O 37
Ignore - support for rejecting files 37
KeepAlive - support for client-side disconnection 37
MapApi - logic for view mappings 38
Options - parse and store command line options 38
ServerHelperApi - helper methods for creating personal servers 38
Signaler - interrupt handling 38
StrBuf - string manipulation 38
StrDict - field/value manipulation 39
StrNum - small numeric strings 39
StrOps - string operations 39
StrPtr - text operations 39
StrRef - refer to existing strings 40
MonItems - between real-time counters and storage 40
MonItem - base class for performance counters 41
MonInteger - store 64-bit integer values 42
MonIntMax - tracks the highest recorded value 42

ClientApi - Helix server connections and commands

The `ClientApi` class represents a connection with the Helix server.

Member functions in this class are used to establish and terminate the connection with the server, establish the settings and protocols to use while running commands, and run Helix server commands over the connection.

I/O is handled by a `ClientUser` object, and errors are captured in an `Error` object. A `ClientApi` object maintains information about client-side settings (`P4PORT`, etc.) and protocol information, such as the server version, and whether "tagged" output is enabled.

`ClientApi` does not include any virtual functions, and typically does not need to be subclassed.

Any Helix server command that is executed must be invoked through `ClientApi::Run()` after first opening a connection using `ClientApi::Init()`. A single connection can be used to invoke multiple commands by calling `Run()` multiple times after a single `Init()`; this approach provides faster performance than using multiple connections.

ClientProgress - progress indicators for Helix server commands

The `ClientProgress` class introduced in 2012.2 provides a means to report on the progress of running commands; you can customize this behavior by subclassing `ClientUser` and `ClientProgress`.

In `ClientUser`, implement `ClientUser::CreateProgress()` and `ClientUser::ProgressIndicator()`. In `ClientProgress`, implement `ClientProgress::Description()`, `ClientProgress::Total()`, `ClientProgress::Update()`, and `ClientProgress::Done()`

The methods of your `ClientProgress` object will be called during the life of a server command. Usually, `Description()` is called first with a `description` and a `units` from the server; the units of measure apply to the `Total()` and `Update()` methods. `Total()` is called if a there is a known upper bound to the number of units, while `Update()` is called from time to time as progress is made. If your `Update()` implementation returns non-zero, the API assumes the user has also attempted to cancel the operation. `Done()` is called last, with the `fail` argument being non-zero in case of failure. When the command is complete, the API destroys the object by calling the destructor.

Default implementations are used in the `p4` command-line client, and report on the progress of `p4 -I submit` and `p4 -I sync -q`.

ClientUser - I/O for Helix server commands

The `ClientUser` class is used for all client-side input and output. This class implements methods that return output from the server to the user after a command is invoked, and gather input from the user when needed.

Member functions in this class are used to format and display server output, invoke external programs (such as text editors, diff tools, and merge tools), gather input for processing by the server, and to handle errors.

Customized functionality in a Helix server application is most typically implemented by subclassing `ClientUser`. In order to enable such customization, nearly all of `ClientUser`'s methods are virtual. The default implementations are used in the `p4` command-line client.

Error - collect and report layered errors

Member functions in this class are used to store error messages, along with information about generic type and severity, format error messages into a form suitable for display to an end user, or marshal them into a form suitable for transferring over a network.

`Error` objects are used to collect information about errors that occur while running a Helix server command.

When a connection is opened with `ClientApi::Init()`, a reference to an `Error` object is passed as an argument to `Init()`. This `Error` object then accumulates any errors that occur; a single `Error` object can hold information about multiple errors. The `Error` can then be checked, and its contents reported if necessary.

Although `Error` itself does not provide any virtual methods that can be re-implemented, the manner in which errors are handled can be changed by re-implementing `ClientUser::HandleError()`. The default behavior for handling errors typically consists of simply formatting and displaying the messages, but `Error` objects maintain additional information, such as severity levels, which can be used to handle errors more intelligently.

ErrorLog - output error messages

The `ErrorLog` class is used to report layered errors, either by displaying error messages to `stderr`, or by redirecting them to logfiles. On UNIX systems, error messages can also be directed to the `syslog` daemon.

FileSys - Helix server file I/O

The `FileSys` class provides a platform-independent set of methods used to create, read and write files to disk.

You can intercept the file I/O and implement your own client workspace file access routines by replacing `FileSys *ClientUser::File()` in a `ClientUser` subclass.

> **Note**
> Replacing the existing I/O routines is non-trivial. Your replacement routines must handle all special cases, including cross-platform file issues.
>
> Unless your application has highly specialized requirements, (for instance, performing all file I/O in memory rather than on disk), this approach is not recommended.

If you intend to replace `File()`, all of the virtual methods documented are required. The non virtual methods are not required and not documented.

Ignore - support for rejecting files

The `Ignore` class has two methods, `Ignore::Reject()` and `Ignore::RejectCheck()`. Both methods are used by applications to determine whether files destined to be opened for add will be rejected due to matching an entry in an ignore files.

KeepAlive - support for client-side disconnection

The `KeepAlive` class has only one method, `KeepAlive::IsAlive()`. The method is used by applications to support client-side command termination.

MapApi - logic for view mappings

The `MapApi` class allows a client application to duplicate the logic used by the server when interpreting and combining view mappings such as branch views, client views, and protections.

Each `MapApi` object represents a single mapping that is built by calling `MapApi::Insert()` to add new lines. A file can be translated through the mapping or tested for inclusion by calling `MapApi::Translate()`. Two `MapApi` objects may be combined into a single new `MapApi` object (for example, a client view and a protection table may be joined into a single mapping that represents all files in the client view that are included in the protection table) by calling `MapApi::Join()`.

Options - parse and store command line options

The `Options` class encapsulates functions useful for parsing command line flags, and also provides a means of storing flag values.

Sample code is provided to illustrate how `Options::GetValue()` and `Options::Parse()` work together to parse command line options.

ServerHelperApi - helper methods for creating personal servers

The `ServerHelperApi` class is used to create personal servers. This class is only necessary when initially creating a local server; once it is created you can use the standard `ClientApi` and `ClientUser` classes to operate against the server.

Signaler - interrupt handling

The `Signaler` class enables the API programmer to register functions that are to be called when the client application receives an interrupt signal. The `Signaler` class maintains a list of registered functions and calls each one in turn.

By default, after all of the registered functions have been executed, the process exits, returning -1 to the operating system.

StrBuf - string manipulation

The `StrBuf` class is the preferred general string manipulation class. This class manages the memory associated with a string, including allocating new memory or freeing old memory as required.

The `StrBuf` class is derived from the `StrPtr` class, and makes heavy use of the `buffer` and `length` members inherited from the `StrPtr` class. The `buffer` member of a `StrBuf` instance is a pointer to the first byte in the string. The `length` member of a `StrBuf` instance is the length of the string.

Most member functions maintain the string pointed to by the `buffer` member of a `StrBuf` as a null-terminated string. However, the `Clear` member function does not set the first byte of the string to a null byte, nor does the `Extend` member function append a null byte to an extended string. If you need to maintain a string as null-terminated when using the `Clear()` and `Extend()` member functions, follow the calls to `Clear()` and `Extend()` with calls to `Terminate()`.

A number of member functions move the string pointed to by a `StrBuf`'s buffer, and change the `buffer` member to point to the new location. For this reason, do not cache the pointer. Use `StrPtr::Text()` whenever the pointer a `StrBuf`'s buffer is required.

StrDict - field/value manipulation

The `StrDict` class provides a dictionary object of `StrPtr` s with a simple Get/Put interface. This class contains abstract methods and therefore cannot be instantiated, but its subclasses adhere to the basic interface documented here.

`ClientApi` is a descendant of `StrDict`; most notably, the `StrDict::SetArgv()` method is used to set the arguments to a Helix server command before executing it with `ClientApi::Run()`.

The `ClientUser::OutputStat()` method takes a `StrDict` as an argument; the `StrDict` methods are therefore necessary to process data with `OutputStat()`. Note that pulling information from a `StrDict` is typically easier than trying to parse the text given to `OutputInfo()`.

StrNum - small numeric strings

The `StrNum` class, derived from `StrPtr`, is designed to hold a small string representing a number. Like a `StrBuf`, it handles its own memory. Unlike a `StrBuf`, it does not dynamically resize itself, and is limited to 24 characters, meaning that the largest number that can be represented by a `StrNum` is 999999999999999999999999.

StrOps - string operations

`StrOps` is a memberless class containing static methods for performing operations on strings.

StrPtr - text operations

The `StrPtr` class is a very basic pointer/length pair used to represent text.

This class provides a number of methods for comparison and reporting, but it is not in itself very useful for storing data; the `StrBuf` child class is a more practical means of storing data, as it manages its own memory.

StrRef - refer to existing strings

The `StrRef` class is a simple pointer/length pair representing a string. The `StrRef` class is is derived from `StrPtr` and does not add a great deal of new functionality to that class, with the exception of methods that make the pointer mutable (and therefore usable), whereas a base `StrPtr` is read-only.

As its name suggests, a `StrRef` serves as a reference to existing data, as the class does not perform its own memory allocation. The `StrBuf` class is most useful when storing and manipulating existing strings.

MonItems - between real-time counters and storage

The **MonItems** class is the bridge between the counters and the storage.

The **MemItems** class either maps the contents of an existing memory-mapped file into **MonItem** objects or maps registered **MonItem** objects into a new or existing memory-mapped file.

If you want to write a utility that accesses the counters and reads their values, see the example that exports the counters in a manner compatible with Prometheus real-time monitoring, `p4mon-prometheus-exporter.cc`, which is located inside `http://ftp.perforce.com/perforce/r21.1/bin.tools/p4source.tgz`

For more information, see p4 monitor realtime in *Helix Core P4 Command Reference*.

Constructor
`MonItems()`

Methods
`bool Load(const char *file, Error *e)` Loads an existing memory mapped file, registering the `MonItem` objects in the file
`void Display()` Dumps all the registered `MonItem` objects to STDOUT
`MonItem *First()` Returns the first registered `MonItem` object. Use with `Next(MonItem *counter)` to iterate over `MonItem` objects
`MonItem *Next(MonItem *counter)` Returns the next registered `MonItem` object. Use with `First()` to iterate over `MonItem` objects

MonItem - base class for performance counters

The **MonItem** class contains the infrastructure for mapping counters between variables in code, their registered **MonItem** forms, and the memory representation that will be written to, and read from, the memory-mapped file.

Enums	Values
`MonItemFlags` `MonItem` modifiers (bit mapped)	`MI_HIDDEN (1)` - Hidden counter: used for internal only use, not a performance counter. `MI_CUMULATIVE(2)` - Cumulative counter: values will always increase.

`MonItemTypes`
`MonItem` subclass type
	`MIT_NONE (1)` - A plain `MonItem`
	`MIT_INT (2)` - A `MonInteger`
	`MIT_INTMAX (3)` - A `MonIntMax`

Constructors

`MonItem( const char *name, int flags = 0 )`

Constructs a new `MonItem` with the provided name and flags (defaulting to none) or type `MIT_NONE (1)`

Should only be called by a constructor in a subclass

`MonItem( const MonItem & )`

Default copy constructor

Static Method

`MonItem *Find(const char *name)`

Finds a `MonItem` in the registry by name or returns `NULL` if no matches are found

Operator

`MonItem &operator =( const MonItem & )`

Copy a `MonItem`

Methods

`StrRef Name() const`

Returns the name of the `MonItem` object

Methods
`int Flags() const` Returns the `MonItemFlags` of the `MonItem` object
`bool Active() const` Returns `true` if the `MonItem` is mapped to shared memory

Virtual Methods
`int DataSize()` const The number of bytes used to store the data of the `MonItem` type
`int Alignment() const` The number of bytes to use for alignment padding.
`int ItemId() const` The MonItemTypes of the `MonItem` type
`void Display( StrBuf * ) const` Renders the data of the `MonItem` object as a string

MonInteger - store 64-bit integer values

The **MonInteger** subclass of `MonItem` is used to access 64-bit integer values.

Constructor
`MonInteger( const char *name, int flags = 0 )` Constructs a new `MonInteger` with the provided name and flags (defaulting to none) or type `MIT_INT (2)`

For inherited methods, see "MonItem - base class for performance counters" on the previous page.

Method
`P4INT64 Value() const` Reads the value of the counter

MonIntMax - tracks the highest recorded value

The **MonIntMax** class is a subclass of `MonInteger`. Like **MonInteger**, **MonIntMax** stores 64-bit integer values but also tracks the highest recorded value.

The **MonItem** subclass of `MonItem` is used to access 64-bit integer values.

Constructor

`MonIntMax( const char *name, int flags = 0 )`

Constructs a new `MonIntMax` with the provided name and flags (defaulting to none) or type `MIT_INTMAX (3)`

For inherited methods, see "MonInteger - store 64-bit integer values" on the previous page.

Method

`P4INT64 MaxValue() const`

Reads the highest recorded value of the counter

Public Methods Reference

This section provides information on the available public methods.

ClientApi methods **49**
ClientApi::DefineClient(const char *, Error *) 49
ClientApi::DefineHost(const char *, Error *) 50
ClientApi::DefineIgnoreFile(const char *, Error *) 51
ClientApi::DefinePassword(const char *, Error *) 51
ClientApi::DefinePort(const char *, Error *) 52
ClientApi::DefineUser(const char *, Error *) 53
ClientApi::Dropped() 54
ClientApi::Final(Error *) 55
ClientApi::GetClient() 55
ClientApi::GetConfig() 56
ClientApi::GetCwd() 57
ClientApi::GetHost() 58
ClientApi::GetIgnore() 59
ClientApi::GetIgnoreFile() 59
ClientApi::GetOs() 60
ClientApi::GetPassword() 61
ClientApi::GetPort() 62
ClientApi::GetProtocol(const char *) 63
ClientApi::GetTrans( ) 64
ClientApi::GetUser() 66
ClientApi::Init(Error *) 66
ClientApi::Run(const char *, ClientUser *) 67
ClientApi::SetBreak(KeepAlive *breakCallback) 68
ClientApi::SetClient(const StrPtr *) 70
ClientApi::SetClient(const char *) 71
ClientApi::SetCwd(const StrPtr *) 71
ClientApi::SetCwd(const char *) 72
ClientApi::SetCwdNoReload(const StrPtr *) 73
ClientApi::SetCwdNoReload(const char *) 74
ClientApi::SetHost(const StrPtr *) 75
ClientApi::SetHost(const char *) 75
ClientApi::SetIgnoreFile(const StrPtr *) 76
ClientApi::SetIgnoreFile(const char *) 77
ClientApi::SetPassword(const StrPtr *) 78
ClientApi::SetPassword(const char *) 79
ClientApi::SetPort(const StrPtr *) 79
ClientApi::SetPort(const char *) 80
ClientApi::SetProg(const StrPtr *) 80
ClientApi::SetProg(const char *) 81
ClientApi::SetProtocol(char *, char *) 82
ClientApi::SetProtocolV(char *) 85
ClientApi::SetTicketFile(const StrPtr *) 85

ClientApi::SetTicketFile(const char *) ... 86
ClientApi::SetTrans(int value) ... 87
ClientApi::SetTrustFile(const char *c) ... 87
ClientApi::SetTrustFile(const StrPtr *c) ... 88
ClientApi::SetUi(ClientUser *) ... 88
ClientApi::SetUser(const StrPtr *) ... 89
ClientApi::SetUser(const char *) ... 90
ClientApi::SetVersion(const StrPtr *) ... 91
ClientApi::SetVersion(const char *) ... 92
ClientProgress methods ... 93
ClientProgress::Description(const StrPtr *, int) ... 93
ClientProgress::Done(int) ... 94
ClientProgress::Total(long) ... 95
ClientProgress::Update(long) ... 96
ClientUser methods ... 96
ClientUser::CreateProgress(int) ... 97
ClientUser::Diff(FileSys *, FileSys *, int, char *, Error *) ... 97
ClientUser::Diff(FileSys *, FileSys *, FileSys *, int, char *, Error *) ... 100
ClientUser::Edit(FileSys *, Error *) ... 100
ClientUser::ErrorPause(char *, Error *) ... 101
ClientUser::File(FileSysType) ... 102
ClientUser::Finished() ... 103
ClientUser::HandleError(Error *) ... 104
ClientUser::HandleUrl( ) ... 104
ClientUser::Help(const char *const *) ... 105
ClientUser::InputData(StrBuf *, Error *) ... 106
ClientUser::Merge(FileSys *, FileSys *, FileSys *, FileSys *, Error *) ... 107
ClientUser::Message(Error *) ... 109
ClientUser::OutputBinary(const char *, int) ... 110
ClientUser::OutputError(const char *) ... 110
ClientUser::OutputInfo(char, const char *) ... 111
ClientUser::OutputStat(StrDict *) ... 113
ClientUser::OutputStatPartial( ) ... 115
ClientUser::OutputText(const char *, int) ... 115
ClientUser::ProgressIndicator() ... 116
ClientUser::Prompt(const StrPtr &, StrBuf &, int, Error *) ... 117
ClientUser::RunCmd(const char *, const char *, […], Error *) ... 118
Error methods ... 119
Error::Clear() ... 119
Error::Dump(const char *) ... 120
Error::Fmt(StrBuf *) ... 121
Error::Fmt(StrBuf *, int) ... 122
Error::GetGeneric() ... 123
Error::GetSeverity() ... 124
Error::IsFatal() ... 124
Error::IsWarning() ... 125
Error::Net(const char *, const char *) ... 126
Error::operator << (int) ... 126

Error::operator << (char *) ... 127
Error::operator << (const StrPtr &) ... 127
Error::operator = (Error &) ... 128
Error::Set(enum ErrorSeverity, const char *) ... 128
Error::Set(ErrorId &) ... 129
Error::Sys(const char *, const char *) ... 129
Error::Test() ... 130
ErrorLog methods ... 130
ErrorLog::Abort() ... 130
ErrorLog::Report() ... 131
ErrorLog::SetLog(const char *) ... 132
ErrorLog::SetSyslog() ... 132
ErrorLog::SetTag(const char *) ... 133
ErrorLog::UnsetSyslog() ... 134
FileSys methods ... 134
FileSys::Chmod(FilePerm, Error *) ... 134
FileSys::Close(Error *) ... 136
FileSys::Create(FileSysType) ... 137
FileSys::Open(FileOpenMode, Error *) ... 137
FileSys::Read(const char *, int, Error *) ... 139
FileSys::Rename(FileSys *, Error *) ... 140
FileSys::Set(const StrPtr *) ... 141
FileSys::Stat() ... 142
FileSys::StatModTime() ... 144
FileSys::Truncate() ... 145
FileSys::Unlink(Error *) ... 145
FileSys::Write(const char *, int, Error *) ... 146
Ignore methods ... 147
Ignore::Reject(const StrPtr &, const StrPtr &) ... 147
Ignore::RejectCheck(const StrPtr &) ... 148
KeepAlive methods ... 149
KeepAlive::IsAlive() ... 150
MapApi methods ... 151
MapApi::Clear() ... 151
MapApi::Count() ... 152
MapApi::GetLeft(int) ... 153
MapApi::GetRight(int) ... 153
MapApi::GetType(int) ... 154
MapApi::Insert(const StrPtr &, MapType) ... 155
MapApi::Insert(const StrPtr &, const StrPtr &, MapType) ... 155
MapApi::Join(MapApi *, MapApi *) [static] ... 156
MapApi::Join(MapApi *, MapDir, MapApi *, MapDir) [static] ... 158
MapApi::Translate(const StrPtr &, StrBuf&, MapDir) ... 159
Options methods ... 160
Options::GetValue(char opt, int subopt) ... 160
Options::operator[](char opt) ... 162
Options::Parse(int &, char ** &, const char *, int, const ErrorId &, Error *) ... 164
Options::Parse(int &, StrPtr * &, const char *, int, const ErrorId &, Error *) ... 171

ServerHelperApi methods **171**
ServerHelperApi::ClearProtocol() 172
ServerHelperApi::CloneFromRemote(int,const StrPtr*,ClientUser*,Error*) 172
ServerHelperApi::CloneFromRemote(int,const char*,ClientUser*,Error*) 174
ServerHelperApi::CopyConfiguration(ServerHelperApi*,ClientUser*,Error*) 174
ServerHelperApi::Exists(ClientUser*,Error*) 175
ServerHelperApi::GetCaseFlag() 176
ServerHelperApi::GetClient() 176
ServerHelperApi::GetClient(Error*) 176
ServerHelperApi::GetDvcsDir() 177
ServerHelperApi::GetPort() 178
ServerHelperApi::GetProg() 178
ServerHelperApi::GetQuiet() 179
ServerHelperApi::GetUnicode() 179
ServerHelperApi::GetUser() 179
ServerHelperApi::GetVersion() 180
ServerHelperApi::InitLocalServer(ClientUser*,Error*) 180
ServerHelperApi::PrepareToCloneFilepath(ServerHelperApi*,const char*,ClientUser*,Error*) 182
ServerHelperApi::PrepareToCloneFilepath(ServerHelperApi*,const StrPtr*,ClientUser*,Error*) 183
ServerHelperApi::PrepareToCloneRemote(ServerHelperApi*,const char*,ClientUser*,Error*) 183
ServerHelperApi::PrepareToCloneRemote(ServerHelperApi*,const StrPtr*,ClientUser*,Error*) 184
ServerHelperApi::SetCaseFlag(const char*,Error*) 185
ServerHelperApi::SetCaseFlag(const StrPtr*,Error*) 185
ServerHelperApi::SetClient(const char*) 186
ServerHelperApi::SetClient(const StrPtr*) 186
ServerHelperApi::SetDefaultStream(const char*) 186
ServerHelperApi::SetDefaultStream(const StrPtr*) 187
ServerHelperApi::SetDvcsDir(const char*) 187
ServerHelperApi::SetDvcsDir(const StrPtr*) 188
ServerHelperApi::SetPort(const char*) 188
ServerHelperApi::SetPort(const StrPtr*) 188
ServerHelperApi::SetProg(const char*) 189
ServerHelperApi::SetProg(const StrPtr*) 189
ServerHelperApi::SetProtocol(const char*,const char*) 190
ServerHelperApi::SetProtocolV(const char*) 190
ServerHelperApi::SetQuiet() 190
ServerHelperApi::SetUnicode(int) 191
ServerHelperApi::SetUser(const char*) 191
ServerHelperApi::SetUser(const StrPtr*) 191
ServerHelperApi::SetVersion(const char*) 192
ServerHelperApi::SetVersion(const StrPtr*) 192
Signaler methods **193**
Signaler::Block() 193
Signaler::Catch() 194
Signaler::DeleteOnIntr(void *) 195
Signaler::Intr() 197

Signaler::OnIntr(SignalFunc, void *) 200
Signaler::Signaler() (constructor) 201
StrBuf methods 202
StrBuf::Alloc(int) 202
StrBuf::Append(const char *) 204
StrBuf::Append(const char *, int) 205
StrBuf::Append(const StrPtr *) 207
StrBuf::Clear() 209
StrBuf::StrBuf() (Constructor) 210
StrBuf::StrBuf(const StrBuf &) (Copy Constructor) 211
StrBuf::~StrBuf() (Destructor) 212
StrBuf::Extend(char) 213
StrBuf::Extend(const char *, int) 215
StrBuf::operator =(const char *) 216
StrBuf::operator =(const StrBuf &) 217
StrBuf::operator =(const StrPtr &) 218
StrBuf::operator =(const StrRef &) 219
StrBuf::operator <<(const char *) 220
StrBuf::operator <<(int) 221
StrBuf::operator <<(const StrPtr *) 223
StrBuf::operator <<(const StrPtr &) 224
StrBuf::Set(const char *) 226
StrBuf::Set(const char *, int) 227
StrBuf::Set(const StrPtr *) 228
StrBuf::Set(const StrPtr &) 229
StrBuf::StringInit() 230
StrBuf::Terminate() 233
StrDict methods 235
StrDict::GetVar(const StrPtr &) 235
StrDict::GetVar(const char *) 237
StrDict::GetVar(const char *, Error *) 237
StrDict::GetVar(const StrPtr &, int) 238
StrDict::GetVar(const StrPtr &, int, int) 238
StrDict::GetVar(int, StrPtr &, StrPtr &) 239
StrDict::Load(FILE *) 239
StrDict::Save(FILE *) 241
StrDict::SetArgv(int, char *const *) 241
StrNum methods 243
StrNum::StrNum(int) (constructor) 243
StrNum::Set(int) 243
StrOps methods 244
StrOps::Caps(StrBuf &) 244
StrOps::Dump(const StrPtr &) 245
StrOps::Expand(StrBuf &, StrPtr &, StrDict &) 246
StrOps::Expand2(StrBuf &, StrPtr &, StrDict &) 247
StrOps::Indent(StrBuf &, const StrPtr &) 249
StrOps::Lines(StrBuf &, char *[], int) 250
StrOps::Lower(StrBuf &) 251

StrOps::OtoX(const unsigned char *, int, StrBuf &) 252
StrOps::Replace(StrBuf &, const StrPtr &, const StrPtr &, const StrPtr &) 253
StrOps::Sub(StrPtr &, char, char) 254
StrOps::Upper(StrBuf &) 255
StrOps::Words(StrBuf &, const char *[], char *[], int) 256
StrOps::XtoO(char *, unsigned char *, int) 257
StrPtr methods 258
StrPtr::Atoi() 258
StrPtr::CCompare(const StrPtr &) 259
StrPtr::Compare(const StrPtr &) 260
StrPtr::Contains(const StrPtr &) 261
StrPtr::Length() 262
StrPtr::operator [](int) 263
StrPtr::operators ==, !=, >, <, <=, >= (const char *) 264
StrPtr::operators ==, !=, >, <, <=, >= (const StrPtr &) 265
StrPtr::Text() 266
StrPtr::Value() 267
StrPtr::XCompare(const StrPtr &) 267
StrRef methods 269
StrRef::StrRef() (constructor) 269
StrRef::StrRef(const StrPtr &) (constructor) 269
StrRef::StrRef(const char *) (constructor) 270
StrRef::StrRef(const char * , int) (constructor) 271
StrRef::Null() 272
StrRef::operator =(StrPtr &) 273
StrRef::operator =(char *) 274
StrRef::operator +=(int) 275
StrRef::Set(char *) 276
StrRef::Set(char * , int) 277
StrRef::Set(const StrPtr *) 278
StrRef::Set(const StrPtr &) 279

ClientApi methods

This section lists the available ClientApi methods.

*ClientApi::DefineClient(const char *, Error *)*

Sets `P4CLIENT` in the Windows registry and applies the setting immediately.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new `P4CLIENT` setting

	`Error *e`	an `Error` object
Returns	`void`	

Notes

To make the new `P4CLIENT` setting apply to the next command executed with `Run()`, `DefineClient()` sets the value in the registry and then calls `SetClient()`.

Example

The following code illustrates how this method might be used to make a Windows client application start up with a default `P4CLIENT` setting.

```
client.Init( &e );
client.DefineClient( "default_workspace", &e );
```

*ClientApi::DefineHost(const char *, Error *)*

Sets `P4HOST` in the Windows registry and applies the setting immediately.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new `P4HOST` setting
	`Error *e`	an `Error` object
Returns	`void`	

Notes

To make the new `P4HOST` setting apply to the next command executed with `Run()`, `DefineHost()` sets the value in the registry and then calls `SetHost()`.

Example

The following code illustrates how this method might be used to make a Windows client application start up with a default `P4HOST` setting.

```
client.Init( &e );
client.DefineHost( "default_host", &e );
```

ClientApi::DefineIgnoreFile(const char *, Error *)

Sets `P4IGNORE` in the Windows registry and applies the setting immediately.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new `P4IGNORE` setting
	`Error *e`	an `Error` object
Returns	`void`	

Notes

To make the new `P4IGNORE` setting apply to the next command executed with `Run()`, `DefineIgnoreFile()` sets the value in the registry and then calls `SetIgnoreFile()`.

See also

ClientApi::GetIgnore() ClientApi::GetIgnoreFile() ClientApi::SetIgnoreFile()

Example

The following code illustrates how this method might be used to make a Windows client application start up with a default `P4IGNORE` setting.

```
# include "clientapi.h"

int main()
{
    ClientApi client;
    Error e;

    client.Init( &e );
    client.DefineIgnoreFile( ".p4ignore", &e );
}
```

ClientApi::DefinePassword(const char *, Error *)

Sets `P4PASSWD` in the Windows registry and applies the setting immediately.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new `P4PASSWD` setting
	`Error *e`	an `Error` object
Returns	`void`	

Notes

To make the new `P4PASSWD` setting apply to the next command executed with `Run()`, `DefinePassword()` sets the value in the registry and then calls `SetPassword()`.

`DefinePassword()` does *not* define a new server-side password for the user.

Call `DefinePassword()` with either the plaintext password, or its MD5 hash

Example

The following code illustrates how this method might be used to make a Windows client application start up with a default `P4PASSWD` setting.

```
client.Init( &e );
client.DefinePassword( "default_pass", &e );
```

*ClientApi::DefinePort(const char *, Error *)*

Sets `P4PORT` in the Windows registry and applies the setting immediately.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new `P4PORT` setting
	`Error *e`	an `Error` object
Returns	`void`	

Notes

In order to make the new `P4PORT` setting apply to the next client connection opened with `Init()`, `DefinePort()` sets the value in the registry and then calls `SetPort()`.

Example

The following code illustrates how this method might be used to make a Windows client application automatically set itself to access a backup server if the primary server fails to respond. (This example assumes the existence of a backup server that perfectly mirrors the primary server.)

```
client.Init( &e );

if ( e.IsFatal() )
{
    e.Clear();
    ui.OutputError( "No response from server - switching to backup!\n"
);
    client.DefinePort( "backup:1666", &e );
    client.Init( &e );
}
```

The first command to which the primary server fails to respond results in the error message and the program reinitializing the client to point to the server at `backup:1666`. Subsequent commands do not display the warning because the new `P4PORT` value has been set in the registry.

*ClientApi::DefineUser(const char *, Error *)*

Sets `P4USER` in the Windows registry and applies the setting immediately.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new `P4USER` setting
	`Error *e`	an `Error` object
Returns	`void`	

Notes

To make the new `P4USER` setting apply to the next command executed with `Run()`, `DefineUser()` sets the value in the registry and then calls `SetUser()`.

Example

The following code illustrates how this method might be used to make a Windows client application start up with a default `P4USER` setting.

```
client.Init( &e );
client.DefineUser( "default_user", &e );
```

ClientApi::Dropped()

Check if connection is no longer usable.

Virtual?	No	
Class	`ClientApi`	
Arguments	None	
Returns	`int`	nonzero if the connection has dropped

Notes

`Dropped()` is usually called after `Run()`; it then checks whether the command completed successfully. If the `Init()` is only followed by one `Run()`, as in `p4api.cc`, calling `Final()` and then checking the `Error` is sufficient to see whether the connection was dropped. However, if you plan to make many calls to `Run()` after one call to `Init()`, `Dropped()` provides a way to check that the commands are completing without actually cleaning up the connection with `Final()`.

Example

The `Dropped()` method is useful if you want to reuse a client connection multiple times, and need to make sure that the connection is still alive.

For example, an application for stress-testing a Helix server might run "`p4 have`" 10,000 times or until the connection dies:

```
ClientApi client;
MyClientUser ui;  //this ClientUser subclass doesn't output anything.
Error e;

client.Init( &e );
int count = 0;
while ( !( client.Dropped() ) && count < 10000 )
{
    count++;
    client.Run( "have", &ui );
}
```

```
printf( "Checked have list %d times.\n", count );
client.Final( &e ); // Clean up connection.
```

If the `Dropped()` result is true, the `while` loop ends. The actual error message remains inaccessible until after the call to `client.Final()` to close the connection and store the error.

*ClientApi::Final(Error *)*

Close connection and return error count.

Virtual?	No	
Class	`ClientApi`	
Arguments	`Error *e`	an `Error` object
Returns	`int`	final number of errors

Notes

Call this method after you are finished using the `ClientApi` object to clean up the connection. Every call to `Init()` must eventually be followed by exactly one call to `Final()`.

Example

The following example is a slight modification of `p4api.cc`, and reports the number of errors before the program exits:

```
client.Init( &e );

client.SetArgv( argc - 2, argv + 2 );
client.Run( argv[1], &ui );

printf( "There were %d errors.\n", client.Final( &e ) );
```

ClientApi::GetClient()

Get current client setting.

Virtual?	No	
Class	`ClientApi`	
Arguments	None	
Returns	`const StrPtr &`	a reference to the client setting

Notes

The return value of `GetClient()` is a fixed reference to this `ClientApi` object's setting.

Assigning the return value to a `StrPtr` results in a `StrPtr` containing a `Text()` value that changes if the `ClientApi` object's client setting changes.

Assigning the return value to a `StrBuf` copies the text in its entirety for future access, rather than simply storing a reference to data that might change later.

Under some circumstances, `GetClient()` calls `GetHost()` and returns that value - specifically, if no suitable `P4CLIENT` value is available in the environment, or previously set with `SetClient()`. (This is why, under the `p4` client, client name defaults to the host name if not explicitly set.)

In some instances, `GetHost()` does not return valid results until after a call to `Init()` - see the `GetHost()` documentation for details.

Example

This example demonstrates the use of `GetClient()` and the difference between `StrPtr's` and `StrBuf's`.

```
ClientApi client;
StrPtr p;
StrBuf b;

client.Init();
client.SetClient( "one" );
p = client.GetClient();
b = client.GetClient();
client.SetClient( "two" );

printf( "Current client %s = %s\n", client.GetClient().Text(), p.Text
() );
printf( "Previous client setting was %s\n", b.Text() );
```

Executing the preceding code produces the following output:

```
Current client two = two
Previous client setting was one
```

ClientApi::GetConfig()

Get current configuration file.

Virtual?	No	
Class	`ClientApi`	
Arguments	None	
Returns	`const StrPtr &`	a reference to the config file setting

Notes

See `GetClient()` for more about the `StrPtr` return value.

If the `P4CONFIG` has not been set, `GetConfig()` returns “noconfig”.

Example

The following example demonstrates the usage of `GetConfig()`.

```
ClientApi client;


printf( "Current P4CONFIG is %s\n", client.GetConfig().Text() );
```

Executing the preceding code without having specified a configuration file produces the following output:

```
C:\perforce> a.out
Current P4CONFIG is noconfig
```

ClientApi::GetCwd()

Get current working directory.

Virtual?	No	
Class	`ClientApi`	
Arguments	None	
Returns	`const StrPtr &`	a reference to the name of the current directory

Notes

See `GetClient()` for more about the `StrPtr` return value.

If the working directory has been set by a call to `SetCwd()` or `SetCwdNoReload()`, subsequent calls to `GetCwd()` return that setting regardless of the actual working directory.

Example

The following example demonstrates the usage of `GetCwd()`.

```
ClientApi client;

printf( "Current directory is %s\n", client.GetCwd().Text() );
```

Executing the preceding code produces the following output:

```
C:\perforce> a.out
Current directory is c:\perforce
```

ClientApi::GetHost()

Get client hostname.

Virtual?	No	**Class**
`ClientApi`		**Arguments**
None		**Returns**

Notes

See `GetClient()` for more about the `StrPtr` return value.

In some instances, `GetHost()` is not valid until after the network connection has been established with `Init()`. `GetHost()` attempts to pull its value from earlier `SetHost()` calls, then from `P4HOST` in the environment, and then from the value of "hostname" returned by the client OS. If none of these is applicable, a reverse DNS lookup is performed, but the lookup will not work unless the connection has been established with `Init()`.

To guarantee valid results, call `GetHost()` only after `Init()` or `SetHost()`. As `GetHost()` may sometimes be called during the execution of `GetClient()`, this warning applies to both methods.

As noted above, `GetHost()` does not necessarily return the actual hostname of the machine if it has been overridden by `P4HOST` or an earlier call to `SetHost()`.

Example

The following example demonstrates the usage of `GetHost()`.

```
ClientApi client;
client.Init();

printf( "Client hostname is %s\n", client.GetHost().Text() );
```

Executing the preceding code produces the following output:

```
shire% a.out
Client hostname is shire
```

ClientApi::GetIgnore()

Virtual?	No	
Class	`ClientApi`	
Arguments	None	
Returns	`Ignore *i`	an `Ignore` object, which can can be used to determine if a path is ignored.

Notes

If `P4IGNORE` is not set, no paths are ignored.

See also

`ClientApi::DefineIgnoreFile()` `ClientApi::GetIgnoreFile()` `ClientApi::SetIgnoreFile()`

Example

This example demonstrates the use of `GetIgnore()`.

```
if ( client->GetIgnore()->Reject( *clientPath,
                                  client->GetIgnoreFile() ) )
    {
        /* handling for ignored file */
    }
}
```

ClientApi::GetIgnoreFile()

Get the full path name of the ignore file used for the current connection.

Virtual?	No
Class	`ClientApi`

Arguments	None	
Returns	`const StrPtr &`	a reference to the path of the ignore file.

Notes

See `GetClient()` for more about the `StrPtr` return value.

If the `P4IGNORE` is unset, `GetIgnoreFile()` returns an uninitialized `StrPtr`.

See also

`ClientApi::DefineIgnoreFile()` `ClientApi::GetIgnore()` `ClientApi::SetIgnoreFile()`

Example

This example demonstrates the use of `GetIgnoreFile()`.

```
# include "clientapi.h"

int main()
{
    ClientApi client;
    printf( "The current ignore file is '%s'\n",
            client.GetIgnoreFile().Text() );
}
```

Executing the preceding code produces output similar to the following:

```
The current ignore file is .p4ignore
```

ClientApi::GetOs()

Get name of client operating system.

Virtual?	No	
Class	`ClientApi`	
Arguments	None	
Returns	`const StrPtr &`	a reference to the OS string

Notes

See `GetClient()` for more about the `StrPtr` return value.

`GetOs()` returns one of “UNIX”, “vms”, “NT”, “Mac”, or null.

Example

The following example demonstrates the usage of `GetOs()`.

```
ClientApi client;
printf( "Client OS is %s\n", client.GetOs().Text() );
```

Executing the preceding code under Windows produces the following output:

```
C:\perforce> a.out
Client OS is NT
```

Executing the preceding code on a UNIX machine produces the following output:

```
shire$ a.out
Client OS is UNIX
```

ClientApi::GetPassword()

Get password setting.

Virtual?	No	
Class	`ClientApi`	
Arguments	None	
Returns	`const StrPtr &`	a reference to the password

Notes

See `GetClient()` for more about the `StrPtr` return value.

This method returns the password currently set on the client, which may or may not be the one set on the server for this user. The command "`p4 passwd`" sets `P4PASSWD` on the client machine to an MD5 hash of the actual password, in which case `GetPassword()` returns this MD5 hash rather than the plaintext version.

However, if the user sets `P4PASSWD` directly with the plaintext version, `GetPassword()` returns that plaintext version. In both instances, the result is the same as that displayed by "`p4 set`" or an equivalent command that displays the value of the `P4PASSWD` environment variable.

`SetPassword()` overrides the `P4PASSWD` value, and subsequent `GetPassword()` calls return the new value set by `SetPassword()` rather than the one in the environment.

Example

The following example demonstrates the usage of `GetPassword()`.

```
ClientApi client;

printf( "Your password is %s\n", client.GetPassword().Text() );
```

The following session illustrates the effect of password settings on `GetPassword()`:

```
> p4 set P4PASSWD=p455w04d
> a.out
Your password is p455w04d

> p4 passwd
Enter new password:
Re-enter new password:
Password updated.

> a.out
Your password is 6F577E10961C8F7B519501097131787C
```

ClientApi::GetPort()

Get current port setting.

Virtual?	No	
Class	`ClientApi`	
Arguments	None	
Returns	`const StrPtr &`	a reference to the port setting

Notes

See `GetClient()` for more about the `StrPtr` return value.

If the environment variable `P4PORT` is unset, `GetPort()` sets the port to the default value of `perforce:1666`.

Example

The following example demonstrates the usage of `GetPort()`.

```
ClientApi client;

printf( "You're looking for a server at %s\n", \
         client.GetPort().Text() );
```

Executing the preceding code produces the following output:

```
You're looking for a server at perforce:1666
```

*ClientApi::GetProtocol(const char *)*

Get protocol information for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *v`	the name of the protocol variable being checked
Returns	`StrPtr *`	a pointer to the variable's value

Notes

If the variable is unset, the return value is null. If there is a value, it will be a number in most cases, but in the form of a `StrPtr` rather than an `int`.

Call `GetProtocol()` only after a call to `Run()`, because protocol information is not available until after a call to `Run()`. Calling `GetProtocol()` before `Run()` results in a return value of `null`, which looks misleadingly like an indication that the variable is unset.

`GetProtocol()` reports only on variables set by the server, not variables set by the client with calls to `SetProtocol()`.

Example

The following example code checks whether the server is case-sensitive.

```
...
client.Init( &e );
...
client.Run();

if ( client.Dropped() )
{
    client.Final( &e );
```

```
}

if ( client.GetProtocol( "nocase" ) )
    printf( "Server case-insensitive.\n" );
else
    printf( "Server is case-sensitive.\n" );
```

ClientApi::GetTrans( )

Gets an integer that represents the charset.

Virtual?	No	
Class	`ClientApi`	
Arguments	None	
Returns	`charset`	The integer value for the detected charset

Example

```
printf("charset = %d\n", client.GetTrans());
```

Notes

Returns an integer value based on the detected charset:

```
none=0
```

```
utf8=1
```

```
iso8859-1=2
```

```
utf16-nobom=3
```

```
shiftjis=4
```

```
eucjp=5
```

```
winansi=6
```

```
winoem=7
```

```
macosroman=8
```

```
iso8859-15=9
```

```
iso8859-5=10
koi8-r=11
cp1251=12
utf16le=13
utf16be=14
utf16le-bom=15
utf16be-bom=17
utf16=18
utf8-bom=19
utf32-nobom=20
utf32le=21
utf32be=22
utf32le-bom=23
utf32be-bom=24
utf32=25
utf8unchecked=26
utf8unchecked-bom=27
cp949=28
cp936=29
cp950=30
cp850=31
cp858=32
cp1253=33
cp737=34
```

```
iso8859-7=35
```

```
cp1250=36
```

```
cp852=37
```

ClientApi::GetUser()

Get current user setting.

Virtual?	No	
Class	`ClientApi`	
Arguments	None	
Returns	`const StrPtr &`	a reference to the user setting

Notes

See `GetClient()` for more about the `StrPtr` return value.

Example

The following example demonstrates the usage of `GetUser()`.

```
ClientApi client;
printf( "Your username is %s\n", client.GetUser().Text() );
```

Executing the preceding code as `testuser` produces the following output:

```
Your username is testuser
```

ClientApi::Init(Error *)

Establish a connection and prepare to run commands.

Virtual?	No	
Class	`ClientApi`	
Arguments	`Error *e`	an `Error` object
Returns	`void`	

Notes

`Init()` must be called to establish a connection before any commands can be sent to the server. Each call to `Init()` must be followed by exactly one call to `Final()`.

If an error occurs during `Init()`, it is most likely a connection error, with a severity of `E_FATAL`.

Example

The following code from `p4api.cc` opens a connection with `Init()`, sets arguments, runs a command, and closes the connection with `Final()`.

```
ClientUser ui;
ClientApi client;
Error e;

client.Init( &e );

client.SetArgv( argc - 2, argv + 2 );
client.Run( argv[1], &ui );
client.Final( &e );
return 0;
```

*ClientApi::Run(const char *, ClientUser *)*

Run a Helix server command and return when it completes.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *func`	the name of the command to run
	`ClientUser *ui`	a pointer to a `ClientUser` object
Returns	`void`	

Notes

The `func` argument to `Run()` is the Helix server command to run, (for instance, `info` or `files`). Command arguments are not included and must be set separately with `StrDict::SetArgv()`.

Initialize the connection with `Init()` before calling `Run()`, because without a connection, no commands can be sent to the server. Attempting to call `Run()` before `Init()` will probably result in a fatal runtime error.

`Run()` returns only after the command completes. Note that all necessary calls to `ClientUser` methods are made during the execution of `Run()`, as dictated by the server.

> **Tip**
> For managing the network connection of the run, relevant Helix Core server configurables are:
>
> - net.maxwait
> - net.maxwaitclose

Example

The code below runs `p4 info`, using `ClientUser::OutputInfo()` to display the results to the user. If a subclass of `ClientUser` is used here as the `ui` argument, that subclass's implementation of `OutputInfo()` is used to display the results of the command.

```
ClientApi client;
ClientUser ui;
Error e;

client.Init( &e );
client.Run( "info", &ui );
client.Final( &e );
```

*ClientApi::SetBreak(KeepAlive *breakCallback)*

Establish a callback that is called every 0.5 seconds during command execution.

Virtual?	No	
Class	`ClientApi`	
Arguments	`KeepAlive *breakCallback`	keepalive callback for user interrupt
Returns	`void`	

Notes

To establish the callback routine, you must call `SetBreak()` after `ClientApi::Init()`.

See also

`KeepAlive::IsAlive()`

Example

The following example implements a custom **`IsAlive()`** that can be called three times before returning 0 and terminating the connection. If the call to run the **`changes`** command takes less than 1.5 seconds to complete on the server side, the program outputs the list of changes. If the call to run the **`changes`** command takes more than 1.5 seconds, the server process receives an interrupt request.

```
#include <clientapi.h>

// subclass KeepAlive to implement a customized IsAlive function.
class MyKeepAlive : public KeepAlive
{
    public:
    int  IsAlive();
};

// Set up the interrupt callback. After being called 3 times,
// interrupt 3 times, interrupt the current server operation.
int   MyKeepAlive::IsAlive()
{
    static int counter = 0;
    if ( ++counter > 3 )
    {
        counter = 0;
        return( 0 );
    }
    return( 1 );
}

// Now test the callback
ClientUser ui;
ClientApi client;
MyKeepAlive cb;
Error e;

client.Init( &e );
client.SetBreak( &cb );   // SetBreak must happen after the Init
```

```
client.Run( "changes", &ui );
client.Final( &e );
```

*ClientApi::SetClient(const StrPtr *)*

Sets the client setting to be used for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const StrPtr *c`	the new client setting
Returns	`void`	

Notes

`SetClient()` does not permanently set the `P4CLIENT` value in the environment or registry. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following example displays two client specifications by calling `SetClient()` between `Run()` commands.

```
ClientApi client;
ClientUser ui;
StrBuf sb1;
StrBuf sb2;

sb1 = "client_one";
sb2 = "client_two";
args[0] = "-o";

client.SetClient( &sb1 );
client.SetArgv( 1, args );
client.Run( "client", &ui );

client.SetClient( &sb2 );
client.SetArgv( 1, args );
client.Run( "client", &ui );
```

*ClientApi::SetClient(const char *)*

Sets the client setting to be used for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new client setting
Returns	`void`	

Notes

`SetClient()` does not permanently set the `P4CLIENT` value in the environment or registry. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following example displays two client specifications by calling `SetClient()` between `Run()` commands.

```
ClientApi client;
ClientUser ui;

char *args[1];
args[0] = "-o";

client.SetClient( "client_one" );
client.SetArgv( 1, args );
client.Run( "client", &ui );

client.SetClient( "client_two" );
client.SetArgv( 1, args );
client.Run( "client", &ui );
```

*ClientApi::SetCwd(const StrPtr *)*

Sets the working directory to be used for this connection.

Virtual?	No

Class	`ClientApi`	
Arguments	`const StrPtr *c`	the new directory path
Returns	`void`	

Notes

`SetCwd()` does not permanently set a new working directory in the client environment. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following code sets different working directories and displays them with `p4 info`.

```
ClientApi client;
ClientUser ui;
StrBuf sb1;
StrBuf sb2;

sb1 = "C:\one";
sb2 = "C:\two";

client.SetCwd( &sb1 );
client.Run( "info", &ui );

client.SetCwd( &sb2 );
client.Run( "info", &ui );
```

*ClientApi::SetCwd(const char *)*

Sets the working directory to be used for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new directory path
Returns	`void`	

Notes

`SetCwd()` does not permanently set a new working directory in the client environment. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following code sets different working directories and displays them with `p4 info`.

```
ClientApi client;
ClientUser ui;

client.SetCwd( "C:\one" );
client.Run( "info", &ui );

client.SetCwd( "C:\two" );
client.Run( "info", &ui );
```

*ClientApi::SetCwdNoReload(const StrPtr *)*

Sets the working directory to be used for this connection without checking `P4CONFIG`.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const StrPtr *c`	the new directory path
Returns	`void`	

Notes

`SetCwdNoReload()` does not permanently set a new working directory in the client environment. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Unlike `SetCwd()`, `SetCwdNoReload()` ignores any `P4CONFIG` files found in the new directory hierarchy.

Example

The following code sets different working directories and displays them with `p4 info`.

```
ClientApi client;
ClientUser ui;
```

```
StrBuf sb1;
StrBuf sb2;

sb1 = "C:\one";
sb2 = "C:\two";
client.SetCwdNoReload( &sb1 );
client.Run( "info", &ui );

client.SetCwdNoReload( &sb2 );
client.Run( "info", &ui );
```

*ClientApi::SetCwdNoReload(const char *)*

Sets the working directory to be used for this connection without checking `P4CONFIG`.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new directory path
Returns	`void`	

Notes

`SetCwdNoReload()` does not permanently set a new working directory in the client environment. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Unlike `SetCwd()`, `SetCwdNoReload()` ignores any `P4CONFIG` files found in the new directory hierarchy.

Example

The following code sets different working directories and displays them with `p4 info`.

```
ClientApi client;
ClientUser ui;

client.SetCwdNoReload( "C:\one" );
client.Run( "info", &ui );
```

```
client.SetCwdNoReload( "C:\two" );
client.Run( "info", &ui );
```

ClientApi::SetHost(const StrPtr *)

Sets the hostname to be used for this connection.

Virtual?	No	Class
`ClientApi`		**Arguments**
`const StrPtr *c`	the new hostname value	**Returns**

Notes

`SetHost()` does not permanently change the host name of the client or set `P4HOST` in the environment. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following example sets different hostnames and displays them with `p4 info`.

```
ClientApi client;
ClientUser ui;
StrBuf sb1;
StrBuf sb2;

sb1 = "magic";
sb2 = "shire";

client.SetHost( &sb1 );
client.Run( "info", &ui );

client.SetHost( &sb2 );
client.Run( "info", &ui );
```

ClientApi::SetHost(const char *)

Sets the hostname to be used for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new hostname value
Returns	`void`	

Notes

`SetHost()` does not permanently change the host name of the client or set `P4HOST` in the environment. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following example sets different hostnames and displays them with `p4 info`.

```
ClientApi client;
ClientUser ui;

client.SetHost( "magic" );
client.Run( "info", &ui );

client.SetHost( "shire" );
client.Run( "info", &ui );
```

*ClientApi::SetIgnoreFile(const StrPtr *)*

Sets the full path name of the ignore file to be used for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const StrPtr *c`	the full path name of the new ignore file
Returns	`void`	

Notes

`SetIgnoreFile()` does not permanently set the `P4IGNORE` value in the environment or registry. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

See also

`ClientApi::DefineIgnoreFile()` `ClientApi::GetIgnore()` `ClientApi::GetIgnoreFile()`

Example

The following example sets an ignore file location by calling `SetIgnoreFile()`.

```
# include "clientapi.h"

int main()
{
    ClientApi client;
    StrBuf sb;

    sb = ".p4ignore";
    client.SetIgnoreFile( &sb; );
}
```

*ClientApi::SetIgnoreFile(const char *)*

Sets the full path name of the ignore file to be used for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the full path name of the new ignore file
Returns	`void`	

Notes

`SetIgnoreFile()` does not permanently set the `P4IGNORE` value in the environment or registry. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

See also

`ClientApi::DefineIgnoreFile()` > <<clientapi.getignore,`ClientApi::GetIgnore()` `ClientApi::GetIgnoreFile()`

Example

The following example sets a ticket file location by calling `SetIgnoreFile()`.

```
# include "clientapi.h"

int main()
{
    ClientApi client;

    client.SetIgnoreFile( ".p4ignore" );
}
```

*ClientApi::SetPassword(const StrPtr *)*

Sets the password to be used for this connection.

Virtual?	No	**Class**
`ClientApi`		**Arguments**
`const StrPtr *c`	the new password value	**Returns**

Notes

`SetPassword()` does not permanently change the `P4PASSWD` value in the environment, nor does it in any way change the password that has been set on the server. The new setting applies only to authentication attempts for commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following trivial example demonstrates how to hard-code a password into an application without making it (immediately) user-visible.

```
ClientApi client;
ClientUser ui;
StrBuf sb;

sb = "p455w04d";

client.SetPassword( &sb );
client.SetArgv( argc - 2, argv + 2 );
client.Run( argv[1], &ui );
```

*ClientApi::SetPassword(const char *)*

Sets the password to be used for this connection.

Virtual?	No	Class
`ClientApi`		**Arguments**
`const char *c`	the new password value	**Returns**

Notes

`SetPassword()` does not permanently change the `P4PASSWD` value in the environment, nor does it in any way change the password that has been set on the server. The new setting applies only to authentication attempts for commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following trivial example demonstrates how to hard-code a password into an application without making it (immediately) user-visible.

```
ClientApi client;
ClientUser ui;

client.SetPassword( "p455w04d" );
client.SetArgv( argc - 2, argv + 2 );
client.Run( argv[1], &ui );
```

*ClientApi::SetPort(const StrPtr *)*

Sets the port to be used to open this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const StrPtr *c`	the new port value
Returns	`void`	

Notes

`SetPort()` does not permanently change the `P4PORT` value in the environment. The new setting applies only to new connections established by calling this `ClientApi` object's `Init()` method.

Example

The following example demonstrates setting a new port value before initializing the connection.

```
ClientApi client;
Error e;
StrBuf sb;

sb = "ssl:magic:1666";

client.SetPort( &sb );
client.Init( &e );
```

*ClientApi::SetPort(const char *)*

Sets the port to be used to open this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new port value
Returns	`void`	

Notes

`SetPort()` does not permanently change the `P4PORT` value in the environment. The new setting applies only to new connections established by calling this `ClientApi` object's `Init()` method.

Example

The following example demonstrates setting a new port value before initializing the connection.

```
ClientApi client;
Error e;

client.SetPort( "magic:1666" );
client.Init( &e );
```

*ClientApi::SetProg(const StrPtr *)*

Sets the application or script name for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const StrPtr *c`	the new program name
Returns	`void`	

Notes

`SetProg()` sets the identity of a client application as reported by the `p4 monitor` command, or as recorded by server logging.

Call `SetProg()` before calling `Init()`.

See also

`ClientApi::SetVersion()`

Example

The following example appears as `MyApp` in the output of `p4 monitor show`.

```
ClientApi client;
ClientUser ui;
StrBuf sb;
Error e;

sb.Set( "MyApp" );

client.Init( &e );
client.SetProg( &sb );
client.Run( "info", &ui );
```

*ClientApi::SetProg(const char *)*

Sets the application or script name for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new program name
Returns	`void`	

Notes

`SetProg()` sets the identity of a client application as reported by the `p4 monitor` command, or as recorded by server logging.

Call `SetProg()` before calling `Init()`.

See also

`ClientApi::SetVersion()`

Example

The following example appears as `MyApp` in the output of `p4 monitor show`.

```
ClientApi client;
ClientUser ui;
Error e;

client.Init( &e );
client.SetProg( "MyApp" );
client.Run( "info", &ui );
```

*ClientApi::SetProtocol(char *, char *)*

Sets special protocols for the server to use.

Virtual?	No	
Class	`ClientApi`	
Arguments	`char *p`	the name of the variable to set
	`char *v`	the new value for that variable
Returns	`void`	

Notes

`SetProtocol()` must be called before the connection is established with `Init()`.

The following variables are supported by `SetProtocol()`:

Variable	Meaning
`tag`	To enable tagged output (if tagged output for the command is supported by the server), set the `tag` variable to any value.
`specstring`	To enable specially formatted application forms, set the `specstring` to any value.
`api`	Set the `api` variable to the value corresponding to the level of server behavior your application supports.
`enableStreams`	To allow your application to work with stream depots, set the `enableStreams` variable to any value except 'no'. To explicitly disable streams support set `enableStreams` to 'no'.

By default, the value of the `api` protocol variable matches the version of the API with which you built your application. Under most circumstances, you do not need to set the protocol version from within your application. If you are concerned about changes in server behavior, you can manually set the `api` variable in order to protect your code against such changes.

For instance, the "`p4 info`" command supports tagged output as of server release 2003.2, and changes to this format were made in 2004.2. Code requesting tagged output from "`p4 info`" that was compiled against the 2003.1 API library might break (that is, start producing tagged output) when running against a 2003.2 or newer server. To prevent this, set `api` to the value corresponding to the desired server release.

Command	Set api to	Tagged output supported?
`info`	unset	Only if both server and API are at 2004.2 or greater
<=55	Output is not tagged; behaves like 2003.1 or earlier, even if server supports tagged output.	=56
Output is tagged; behaves like 2003.2.	=57	Output is tagged; behaves like 2004.1, 2004.2, or 2005.1.

Tip
See the table that lists the Protocol levels of server and client by server release number in the *Helix Core Server Administrator Guide*.

Example

The following example demonstrates the use of `SetProtocol()` to enable tagged output. The result of this call is that the `ClientUser` object uses `OutputStat()` to handle the output, rather than `OutputInfo()`.

```
ClientApi client;
Error e;

client.SetProtocol( "tag", "" );
client.Init( &e );
client.Run( "branches", &ui );
client.Final( &e );
```

The following code illustrates how to ensure forward compatibility when compiling against newer versions of the Helix C/C++ API or connecting to newer Helix servers.

```
ClientApi client;
Error e;

printf( "Output is tagged depending on API or server level.\n" );
client.SetProtocol( "tag", "" ); // request tagged output
client.Init( &e );
client.Run( "info", &ui );
client.Final( &e );

printf( "Force 2003.1 behavior regardless of API or server level.\n"
);
client.SetProtocol( "tag", "" );   //request tagged output
client.SetProtocol( "api", "55" ); // but force 2003.1 mode (untagged)
client.Init( &e );
client.Run( "info", &ui );
client.Final( &e );

printf( "Request 2003.2 output if API and server support it.\n" );
client.SetProtocol( "tag", "");   // request tagged output
client.SetProtocol( "api", "56"); // force 2003.2 mode (tagged)
client.Init( &e );
client.Run( "info", &ui );
client.Final( &e );
```

The "`p4 info`" command supports tagged output only as of server release 2003.2. In the example, the first `Run()` leaves `api` unset; if both the client API and Helix server support tagged output for `p4 info` (that is, if you link this code with the 2003.2 or later API *and* run it against a 2003.2 or later server), the output is tagged. If you link the same code with the libraries from the 2003.1 release of the API, however, the first `Run()` returns untagged output *even if connected to a 2003.2 server*. By setting `api` to `55`, the second `Run()` ensures 2003.1 behavior regardless of API or server level. The third call to `Run()` supports 2003.2 behavior against a 2003.2 server and protects against future changes.

*ClientApi::SetProtocolV(char *)*

Sets special protocols for the server to use.

Virtual?	No	
Class	`ClientApi`	
Arguments	`char *nv`	the name and value of the variable to set in `var=val` form
Returns	`void`	

Notes

`SetProtocolV()` functions identically to `SetProtocol()`, except that its argument is a single string of the format `variable=value`.

Example

The following example demonstrates the use of `SetProtocolV()` to enable tagged output. The result is that the `ClientUser` object uses `OutputStat()` to handle the output, rather than `OutputInfo()`.

```
ClientApi client;
Error e;

client.SetProtocolV( "tag=" );
client.Init( &e );
client.Run( "branches", &ui );
client.Final( &e );
```

*ClientApi::SetTicketFile(const StrPtr *)*

Sets the full path name of the ticket file to be used for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const StrPtr *c`	the full path name of the new ticket file
Returns	`void`	

Notes

`SetTicketFile()` does not permanently set the `P4TICKETS` value in the environment or registry. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following example sets a ticket file location by calling `SetTicketFile()`.

```
ClientApi client;
StrBuf sb;

sb = "/tmp/ticketfile.txt";
client.SetTicketFile( &sb );
```

*ClientApi::SetTicketFile(const char *)*

Sets the full path name of the ticket file to be used for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the full path name of the new ticket file
Returns	`void`	

Notes

`SetTicketFile()` does not permanently set the `P4TICKETS` value in the environment or registry. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following example sets a ticket file location by calling `SetTicketFile()`.

```
ClientApi client;

client.SetTicketFile( "/tmp/ticketfile.txt" );
```

ClientApi::SetTrans(int value)

Sets Unicode detection.

Virtual?	No	
Class	`ClientApi`	
Arguments	int value	`0` = suppress auto Unicode detection `1` = auto Unicode detection enabled
Returns	void	

Notes

Automatic detection of Unicode mode can be disabled by calling `SetTrans()` prior to calling `Init()`

*ClientApi::SetTrustFile(const char *c)*

Sets the location of the trustfile using the full pathname to the file and not a directory.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the full path name of the new trust file
Returns	`void`	

Notes

`SetTtrustFile()` does not permanently set the `P4TRUST` value in the environment or registry. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following example sets a trust file location by calling `SetTrustFile()`.

```
ClientApi client;
StrBuf sb;

sb = "/tmp/trustfile.txt";
client.SetTrustFile( &sb );
```

*ClientApi::SetTrustFile(const StrPtr *c)*

Sets the location of the trust file using the full pathname to the file and not a directory.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const StrPtr *c`	the full path name of the new trust file
Returns	`void`	

Notes

`SetTtrustFile()` does not permanently set the `P4TRUST` value in the environment or registry. The new setting applies only to commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following example sets a trust file location by calling `SetTrustFile()`.

```
ClientApi client;
StrBuf sb;

sb = "/tmp/trustfile.txt";
client.SetTrustFile( &sb );
```

*ClientApi::SetUi(ClientUser *)*

Reset the `ClientUser` object used for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`ClientUser *ui`	a pointer to a `ClientUser` object.

Returns	`void`

Notes

Unless you pass the `ClientUser` object to the `Run()` method, you must first call `SetUi()`. The new setting applies to commands executed by calling this `ClientApi` object's `Run()` method.

Example

The following example illustrates two ways to run `p4 info`:

```
ClientApi client;
ClientUser ui;

client.Run( "info", &ui );

client.SetUi( &ui );
client.Run( "info" );
```

*ClientApi::SetUser(const StrPtr *)*

Sets the user for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const StrPtr *c`	the new user name setting
Returns	`void`	

Notes

`SetUser()` does not permanently set the `P4USER` value in the environment or registry. Calling this method is equivalent to using the "-u" global option from the command line to set the user value for a single command, with the exception that a single `ClientApi` object can be used to invoke multiple commands in a row.

If the user setting is to be in effect for the command when it is executed, you must call `SetUser()` before calling `Run()`.

Example

The following example displays two user specifications by calling `SetUser()` between `Run()` commands.

```
ClientApi client;
Error e;
StrBuf sb1;
StrBuf sb2;

sb1 = "user1";
sb2 = "user2";

char *args[1];
args[0] = "-o";

client.SetUser( &sb1 );
client.SetArgv( 1, args );
client.Run( "user", &ui );

client.SetUser( &sb2 );
client.SetArgv( 1, args );
client.Run( "user", &ui );
```

*ClientApi::SetUser(const char *)*

Sets the user for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new user name setting
Returns	`void`	

Notes

`SetUser()` does not permanently set the `P4USER` value in the environment or registry. Calling this method is equivalent to using the "-u" global option from the command line to set the user value for a single command, with the exception that a single `ClientApi` object can be used to invoke multiple commands in a row.

If the user setting is to be in effect for the command when it is executed, you must call `SetUser()` before calling `Run()`.

Example

The following example displays two user specifications by calling `SetUser()` between `Run()` commands.

```
ClientApi client;
Error e;

char *args[1];
args[0] = "-o";

client.SetUser( "user1" );
client.SetArgv( 1, args );
client.Run( "user", &ui );

client.SetUser( "user2" );
client.SetArgv( 1, args );
client.Run( "user", &ui );
```

*ClientApi::SetVersion(const StrPtr *)*

Sets the application or script version for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const StrPtr *c`	the new version number
Returns	`void`	

Notes

`SetVersion()` sets the version number of a client application as reported by the `p4 monitor -e` command, or as recorded by server logging.

If a client application compiled with version 2005.2 or later of the API does not call SetVersion(), then the version string reported by p4 monitor -e (and recorded in the server log) defaults to the api value appropriate for the server level as per SetProtocol().

Call `SetVersion()` after calling `Init()` and before each call to `Run()`.

See also

`ClientApi::SetProtocol()` `ClientApi::SetProg()`

Example

The following example appears as `2005.2` in the output of `p4 monitor show -e`.

```
ClientApi client;
ClientUser ui;
StrBuf sb;
Error e;

sb.Set( "2005.2" );

client.Init( &e );
client.SetVersion( &sb );
client.Run( "info", &ui );
```

*ClientApi::SetVersion(const char *)*

Sets the application or script version for this connection.

Virtual?	No	
Class	`ClientApi`	
Arguments	`const char *c`	the new version number
Returns	`void`	

Notes

`SetVersion()` sets the version number of a client application as reported by the `p4 monitor -e` command, or as recorded by server logging.

If a client application compiled with version 2005.2 or later of the API does not call `SetVersion()`, then the version string reported by `p4 monitor -e` (and recorded in the server log) defaults to the pass:[<literal>api</literal> value appropriate for the server level as per `SetProtocol()`.

Call `SetVersion()` after calling `Init()` and before each call to `Run()`.

See also

`ClientApi::SetProtocol()``ClientApi::SetProg()`

Example

The following example appears as `2005.2` in the output of `p4 monitor show -e`.

```
ClientApi client;
ClientUser ui;
Error e;

client.Init( &e );
client.SetVersion( "2005.2" );
client.Run( "info", &ui );
```

ClientProgress methods

This section lists the available ClientProgress methods.

*ClientProgress::Description(const StrPtr *, int)*

Sets up a description and defines the units by which command progress is measured.

Virtual?	Yes	
Class	`ClientProgress`	
Arguments	`const StrPtr *desc`	description from the server
	`int units`	the units in which progress is to be measured
Returns	`void`	

Notes

The API calls this method on command startup, supplying your implementation with a description and a client progress unit type. The `units` in which client progress is measured are defined in `clientprog.h` as follows:

Client Progress Unit	Value	Meaning
`CPU_UNSPECIFIED`	`0`	No units specified
`CPU_PERCENT`	`1`	Value is a percentage
`CPU_FILES`	`2`	Value is a count of files
`CPU_KBYTES`	`3`	Value is in kilobytes
`CPU_MBYTES`	`4`	Value is in megabytes

See also

`ClientUser::CreateProgress()` `ClientUser::ProgressIndicator()` `ClientProgress::Done()` `ClientProgress::Total()` `ClientProgress::Update()`

Example

Create a subclass of `ClientProgress` and define an implementation of `Description()`, even if it is a trivial implementation:

```
void MyProgress::Description( const StrPtr *desc, int units )
{
    printf( "Starting command:\n" );
}
```

ClientProgress::Done(int)

Called when an operation completes.

Virtual?	Yes	
Class	`ClientProgress`	
Arguments	`int fail`	operation status: 1 if failed, 0 if successful
Returns	`void`	

Notes

The API calls `Done()` on command completion with 0 for success, or 1 for failure.

See also

`ClientUser::CreateProgress()` `ClientUser::ProgressIndicator()` `ClientProgress::Description()` `ClientProgress::Total()` `ClientProgress::Update()`

Example

To change the way completed actions are reported, create a subclass of `ClientProgress` and define an alternate implementation of `Done()`. For example, to output "Command failed" or "Command completed" upon success or failure, implement `Done()` as follows:

```
void MyProgress::Done( int fail )
{
```

```
    printf( fail ? "Command failed\n" : "Command completed\n");
}
```

ClientProgress::Total(long)

Defines the number of units requested during the operation, if known.

Virtual?	Yes	
Class	`ClientProgress`	
Arguments	`long units`	Total number of client progress units expected, if known
Returns	`void`	

Notes

The API calls this method if and when it has determined the number of client progress units, as defined by `Description()`, are to be processed during the command.

If the total number of expected units changes during the lifetime of a command, the API may call this method more than once. (The total number of expected units is *not* the same as the number of *remaining* units; certain commands may result in multiple calls to this method as the server determines more about the amount of data to be retrieved.)

See also

`ClientUser::CreateProgress()` `ClientUser::ProgressIndicator()` `ClientProgress::Description()` `ClientProgress::Done()` `ClientProgress::Update()`

Example

To report how many progress units are expected, create a subclass of `ClientProgress` and define an alternate implementation of `Total()`.

For example, the following method outputs the number of units expected and is called when, if, and as the total number of expected units changes over the lifetime of the command:

```
void MyProgress::Total( long units )
{
    printf( "Now expecting %l units\n" );
}
```

ClientProgress::Update(long)

Reports on command progress and can be used for updating progress bars.

Virtual?	Yes	
Class	`ClientProgress`	
Arguments	`long units`	Total number of progress units processed, if known
Returns	`int`	

Notes

The API calls the `Update()` method periodically during the life of a command and reports on the number of client progress units processed. (Because a million calls for an update of one million 1024-byte files would be prohibitive, not every unit of progress is reported.) Instead, the API calls this method periodically depending on a combination of elapsed time and number of client progress units processed.

To request the server cancel a process, see "ClientApi::SetBreak(KeepAlive *breakCallback)" on page 68.

See also

`ClientUser::CreateProgress()` `ClientUser::ProgressIndicator()` `ClientProgress::Description()` `ClientProgress::Done()` `ClientProgress::Total()`

Example

To report on units processed, create a subclass of `ClientProgress` and define an alternate implementation of `Update()`.

```
void MyProgress::Update( long units )
{
    displayGUI( units ); // show how many units have been processed
    return 0; // user has not requested cancel, continue processing
}
```

ClientUser methods

This section lists the available ClientUser methods.

ClientUser::CreateProgress(int)

Create a `ClientProgress` object by subclassing, or null if no progress indicator is desired.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`int ProgressType`	the type of progress to be reported
Returns	`*ClientProgress`	a pointer to the new `ClientProgress` object.

Notes

To enable progress reporting for a command, create a `ClientProgress` object and then implement `ProgressIndicator()` to return 0 or 1 depending on whether or not you want to enable the progress indicator. (You typically implement `ProgressIndicator()` to return 1, and call it only when a progress indicator is desired.)

The API calls this method with the appropriate `ProgressType` as defined in `clientprog.h`. The following `ProgressTypes` can be reported:

Client Progress Type	Value	Meaning
`CPT_SENDFILE`	`1`	Files sent to server
`CPT_RECVFILE`	`2`	Files received from server
`CPT_FILESTRANS`	`3`	Files transmitted
`CPT_COMPUTATION`	`4`	Computation performed server-side

See also

`ClientUser::ProgressIndicator()` `ClientProgress::Description()` `ClientProgress::Done()` `ClientProgress::Total()` `ClientProgress::Update()`

ClientUser::Diff(FileSys *, FileSys *, int, char *, Error *)

Diff two files, and display the results.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`FileSys *f1`	the first file to be diffed

	`FileSys *f2`	the second file to be diffed
	`int doPage`	should output be paged?
	`char *diffFlags`	flags to diff routine
	`Error *e`	an `Error` object
Returns	`void`	

Notes

This method is used by `p4 diff` and to display diffs from an interactive `p4 resolve`. If no external diff program is specified, the diff is carried out with a `Diff` object (part of the Helix C/C++ API); otherwise, `Diff()` simply calls the specified external program.

As with `Merge()`, the external program is invoked with `ClientUser::RunCmd()`.

If `doPage` is nonzero and the `P4PAGER` environment variable is set, the output is piped through the executable specified by `P4PAGER`.

See also

`ClientUser::RunCmd()`

Example

In its default implementation, this method is called by an application when `p4 diff` is run. For example:

```
p4 diff -dc file.c
```

results in a call to `Diff()` with the arguments:

Argument	Value
`f1`	a temp file containing the head revision of depot file `file.c`
`f2`	the local workspace version of file `file.c`
`doPage`	`0`
`diffFlag`	`c`
`e`	a normal `Error` object

The diff is performed by creating a `Diff` object, giving it `f1` and `f2` as its inputs, and -c as its flag. The end result is sent to `stdout`. If either of the files is binary, the message “files differ” is printed instead.

Selecting the “d” option during an interactive `p4 resolve` also calls the `Diff()` method, with the `doPage` argument set to 1.

If the environment variable **P4PAGER** or **PAGER** is set, then setting **doPage** to 1 causes the diff output to be fed through the specified pager. If **P4PAGER** and **PAGER** are unset, **dopage** has no effect and the resolve routine displays the diff output normally.

To enable an application to override the default diff routine, create a subclass of **ClientUser** that overrides the **Diff()** method, and use this subclass in place of **ClientUser**.

As an example, suppose that you have a special diff program designed for handling binary files, and you want **p4 diff** to use it whenever asked to diff binary files (rather than display the default “files differ…”).

Furthermore, you want to keep your current **P4DIFF** setting for the purpose of diffing text files, so you decide to use a new environment variable called **P4DIFFBIN** to reference the binary diff program. If **P4DIFFBIN** is set and one of the files is non-text, the **P4DIFFBIN** program is invoked as **P4DIFF** is in the default implementation. Otherwise, the default implementation is called.

Most of the following code is copied and pasted from the default implementation.

```
MyClientUser::Diff( FileSys *f1, FileSys *f2, int doPage, char *df,
Error *e )
{
    const char *diff = enviro->Get( "P4DIFFBIN" );
    if ( diff && ( !f1->IsTextual() || !f2->IsTextual() ) ) // binary
diff
    {
        if ( !df || !*df )
        {
            RunCmd( diff, 0, f1->Name(), f2->Name(), 0, pager, e );
        }
        else
        {
            StrBuf flags;
            flags.Set( "-", 1 );
            flags << df;
            RunCmd( diff, flags. Text(), f1->Name(), f2->Name(), 0,
pager, e );
        }
    }
    else ClientUser::Diff( f1, f2, doPage, df, e );
}
```

*ClientUser::Diff(FileSys *, FileSys *, FileSys *, int, char *, Error *)*

Diff two files, and output the results to a third file.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`FileSys *f1`	the first file to be diffed
	`FileSys *f2`	the second file to be diffed
	`FileSys *fout`	the target file for diff output
	`int doPage`	should output be paged?
	`char *diffFlags`	flags to diff routine
	`Error *e`	an `Error` object
Returns	`void`	

Notes

This method works like `Diff()`, but instead of sending data to the standard output, writes the data to the specified output file.

*ClientUser::Edit(FileSys *, Error *)*

Bring up the given file in a text editor. Called by all `p4` commands that edit specifications.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`FileSys *f1`	the file to be edited
	`Error *e`	an `Error` object
Returns	`void`	

Notes

The `FileSys *` argument to `Edit()` refers to a client temp file that contains the specification that is to be given to the server. `Edit()` does not send the file to the server; its only job is to modify the file. In the default implementation, `Edit()` does not return until the editor has returned.

There is also a three-argument version of `Edit()`, for which the default two-argument version is simply a wrapper. The three-argument version takes an `Enviro` object as an additional argument, and the two-argument version simply passes the member variable `enviro` as this argument. Only the two-argument version is virtual.

Example

The `p4 client` command is one of several Helix server commands that use `ClientUser::Edit()` to allow the user to modify a specification. When the command is executed, the server sends the client specification to the client machine, where it is held in a temp file. `Edit()` is then called with that file as an argument, and an editor is spawned. When the editor closes, `Edit()` returns, and the temp file is sent to the server.

To allow modification of a specification by other means, such as a customized dialog or an automated process, create a subclass of `ClientUser` that overrides the `Edit()` method and use this subclass in place of `ClientUser`.

Suppose that you have already written a function that takes a `FileSys` as input, opens a custom dialog, and returns when the file has been modified. Replace the body of `Edit()` in your subclass with a call to your function, as follows:

```
void MyClientUser::Edit( FileSys *f1, Error *e )
{
    MyDialog( f1 );
}
```

*ClientUser::ErrorPause(char *, Error *)*

Outputs an error and prompts for a keystroke to continue.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`char *errBuf`	the error message to be printed
	`Error *e`	an `Error` object
Returns	`void`	

Notes

The default implementation of `ErrorPause()` consists solely of calls to `OutputError()` and `Prompt()`.

Example

One situation that results in a call to `ErrorPause()` is an incorrectly edited specification; for example:

```
> p4 client
...
Error in client specification.
Error detected at line 31.
Wrong number of words for field 'Root'.
Hit return to continue...
```

In this instance, the first three lines of output were the `errBuf` argument to `ErrorPause()`; they were displayed using `OutputError()`.

To display an error and prompt for confirmation within a GUI application, create a subclass of `ClientUser` that overrides `ErrorPause()` and use this subclass in place of `ClientUser`.

Suppose that you have a function `MyWarning()` that takes a `char *` as an argument, and displays the argument text in an appropriate popup dialog that has to be clicked to be dismissed. You can implement `ErrorPause()` as a call to this function, as follows:

```
void MyClientUser::ErrorPause( char *errBuf, Error *e )
{
    MyWarning( errBuf );
}
```

Within a GUI, the warning text and "OK" button are probably bundled into a single dialog, so overriding `ErrorPause()` is a better approach than overriding `OutputError()` and `Prompt()` separately.

ClientUser::File(FileSysType)

Create a `FileSys` object for reading and writing files in the client workspace.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`FileSysType type`	the file type of the file to be created
Returns	`FileSys *`	a pointer to the new `FileSys`.

Notes

This method is a wrapper for `FileSys::Create()`.

Example

`ClientUser::File()` is generally called whenever it's necessary to manipulate files in the client workspace. For example, a `p4 sync`, `p4 edit`, or `p4 revert` makes one call to `File()` for each workspace file with which the command interacts.

An alternate implementation might return a subclass of `FileSys`. For example, if you have defined a class `MyFileSys` and want your `MyClientUser` class to use members of this class rather than the base `FileSys`, reimplement `File()` to return a `MyFileSys` instead:

```
FileSys * MyClientUser::File( FileSysType type )
{
    return MyFileSys::Create( type );
}
```

ClientUser::Finished()

Called after client commands finish.

Virtual?	Yes
Class	`ClientUser`
Arguments	None
Returns	`void`

Notes

This function is called by the server at the end of every Helix server command, but in its default implementation, it has no effect. The default implementation of this function is empty - it takes nothing, does nothing, and returns nothing.

Example

To trigger an event after the completion of a command, create a subclass of `ClientUser` and provide a new implementation of `Finished()` that calls that event.

For example, if you want your application to beep after each command, put the command into `Finished()`, as follows.

```
void MyClientUser::Finished()
{
    printf( "Finished!\n%c", 7 );
}
```

ClientUser::HandleError(Error *)

Process error data after a failed command.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`Error *e`	an `Error` object
Returns	`void`	

Notes

The default implementation formats the error with `Error::Fmt()` and outputs the result with `OutputError()`.

2002.1 and newer servers do not call `HandleError()` to display errors. Instead, they call `Message()`. The default implementation of `Message()` calls `HandleError()` if its argument is a genuine error; as a result, older code that uses `HandleError()` can be used with the newer API and newer servers so long as the default implementation of `Message()` is retained.

Example

`HandleError()` is called whenever a command encounters an error. For example:

```
> p4 files nonexistent
nonexistent - no such file(s).
```

In this case, the `Error` object given to `HandleError()` contains the text "nonexistent - no such file (s)." and has a severity of 2 (`E_WARN`).

To handle errors in a different way, create a subclass of `ClientUser` with an alternate implementation of `HandleError()`.

For example, if you want an audible warning on a fatal error, implement `HandleError()` as follows:

```
void MyClientUser::HandleError( Error *err )
{
    if ( err->IsFatal() ) printf ( "Fatal error!\n%c", 7 );
}
```

ClientUser::HandleUrl( )

Allows the application to open a URL other than the URL of the default browser.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`*url`	`const StrPtr`
Returns	`void`	

*ClientUser::Help(const char *const *)*

Displays a block of help text to the user. Used by `p4 resolve` but not `p4 help`.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`const char *const *help`	an array of arrays containing the help text.
Returns	`void`	

Notes

This function is called by `p4 resolve` when the “?” option is selected during an interactive resolve. The default implementation displays the help text given to it, one line at a time.

Example

The default implementation is called in order to display the "merge options" block of help text during a resolve by dumping the text to `stdout`.

To display the resolve help text in another manner, create a subclass of `ClientUser` with an alternate implementation of `Help()`.

For example, suppose you’d like a helpful message about the meaning of "yours" and "theirs" to be attached to the help message. Define the method as follows:

```
void MyClientUser::Help( const char *const *help )
{
    for ( ; *help; help++ )
        printf( "%s\n", *help );
    printf( "Note: In integrations, yours is the target file, \
             theirs is the source file.\n" );
}
```

*ClientUser::InputData(StrBuf *, Error *)*

Provide data from `stdin` to `p4 < command> -i`.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`StrBuf *strbuf`	the `StrBuf` which is to hold the data
	`Error *e`	an `Error` object
Returns	`void`	

Notes

Any command that edits a specification can take the -i option; this method supplies the data for the specification. In the default implementation, the data comes from `stdin`, but an alternate implementation can accept the data from any source. This method is the only way to send a specification to the server without first putting it into a local file.

Example

The default implementation is called during a normal invocation of `p4 client -i`.

```
p4 client -i < clispec.txt
```

In this example, `clispec.txt` is fed to the command as `stdin`. Its contents are appended to the `StrBuf` that is given as an argument to `InputData()`, and this `StrBuf` is given to the server after `InputData()` returns.

To read the data from a different source, create a subclass of `ClientUser` with an alternate implementation of `InputData()`.

For example, suppose that you want to be able to edit a client specification without creating a local temp file. You've already written a function which generates the new client specification and stores it as a `StrBuf` variable in your `ClientUser` subclass. To send your modified client specification to the server when running `p4 client -i` with your modified `ClientUser`, implement `InputData()` to read data from that `StrBuf`.

The example below assumes that the subclass `MyClientUser` has a variable called `mySpec` that already contains the valid client specification before running `p4 client -i`.

```
void MyClientUser::InputData( StrBuf *buf, Error *e )
{
    buf->Set( mySpec );
}
```

*ClientUser::Merge(FileSys *, FileSys *, FileSys *, FileSys *, Error *)*

Call an external merge program to merge three files during resolve.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`FileSys *base`	the "base" file
	`FileSys *leg1`	the "theirs" file
	`FileSys *leg2`	the "yours" file
	`FileSys *result`	the final output file
	`Error *e`	an `Error` object
Returns	`void`	

Notes

`Merge()` is called if the “m” option is selected during an interactive resolve. `Merge()` does not call the merge program; it merely invokes external merge programs (including P4Merge as well as third-party tools). External merge programs must be specified by an environment variable, either `P4MERGE` or `MERGE`. `Merge()` returns after the external merge program exits.

As in `Diff()`, the external program is invoked using `ClientUser::RunCmd()`.

See also

`ClientUser::RunCmd()`

Example

When the "merge" option is selected during an interactive resolve, the file arguments to `Merge()` are as follows:

Argument	Value
`base`	A temp file built from the depot revision that is the "base" of the resolve.
`leg1`	A temp file built from the depot revision that is the "theirs" of the resolve.
`leg2`	The local workspace file that is the "yours" of the resolve.
`result`	A temp file in which to construct the new revision of "yours".

These file arguments correspond exactly to the command-line arguments passed to the merge tool.

After you "accept" the merged file (with “ae”), the "result" temp file is copied into the “leg2” or "yours" workspace file, and this is the file that is submitted to the depot.

To change the way that external merge programs are called during a resolve, create a subclass of **`ClientUser`** with an alternate implementation of **`Merge()`**.

For example, suppose that one of your favorite merge tools, “yourmerge”, requires the “result” file as the first argument. Rather than wrapping the call to the merge tool in a script and requiring your users to set **`P4MERGE`** to point to the script, you might want to provide support for this tool from within your application as follows:

```
void MyClientUser::Merge(
    FileSys *base,
    FileSys *leg1,
    FileSys *leg2,
    FileSys *result,
    Error *e )
{
    char *merger;

    if ( !( merger = enviro->Get( "P4MERGE" ) ) &&
         !( merger = getenv( "MERGE" ) ) )
    {
        e->Set( ErrClient::NoMerger );
        return;
    }

    if ( strcmp( merger, "yourmerge" ) == 0 )
    {
        RunCmd( merger, result->Name(), base->Name(),
                leg1->Name(), leg2->Name(), 0, e );
    }
    else
    {
        RunCmd( merger, base->Name(), leg1->Name(),
                leg2->Name(), result->Name(), 0, e );
    }
}
```

*ClientUser::Message(Error *)*

Output information or errors.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`Error *e`	an `Error` object containing the message
Returns	`void`	

Notes

`Message()` is used by 2002.1 and later servers to display information or errors resulting from Helix server commands. Earlier versions of the Helix server call `OutputInfo()` to display information, and `HandleError()` to display errors.

The default implementation of `Message()` makes calls to `OutputInfo()` or `HandleError()` as appropriate. If you want your application to be compatible with pre-2002.1 servers, use this default implementation of `Message()` - newer servers will call `Message()`, and older servers will call `OutputInfo()` and `HandleError()` directly.

If you re-implement `Message()` to handle errors and information in a different way, be advised that older servers will still call `OutputInfo()` and `HandleError()` rather than your `Message()` method.

Example

```
> p4 files //depot/proj/...
//depot/proj/file.c#1 - add change 456 (text)
```

In this example, the server passes a single `Error` object to the `ClientUser`'s `Message()` method, with a severity of `E_INFO` and text "//depot/proj/file.c#1 - add change 456 (text)". The default `Message()` method detects that this was an "info" message, and passes the text to `OutputInfo()`, which by default sends the text to `stdout`.

To handle messages differently, subclass `ClientUser` and re-implement the `Message()` method (see the preceding note on interoperability with old servers if you do this).

For example, to take all server messages and load them into a `StrBuf` that is a member of your `ClientUser` class, use the following:

```
void MyClientUser::Message( Error *err )
{
    StrBuf buf;
    err->Fmt( buf, EF_PLAIN );
```

```
    myBuf.Append( buf );
}
```

ClientUser::OutputBinary(const char *, int)

Output binary data.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`const char *data`	a pointer to the first byte of data to output
	`int length`	the number of bytes to output
Returns	`void`	

Notes

The default implementation of `OutputBinary()` writes the contents of a binary file to `stdout`. A call to `OutputBinary()` is typically the result of running `p4 print` on a binary file:

```
p4 print //depot/file.jpg > newfile.jpg
```

Example

To modify the way in which binary files are output with `p4 print`, create a subclass of `ClientUser` with an alternate implementation of `OutputBinary()`.

For example, suppose that you want PDF files to be printed to `stdout` as plain text. Add the following code (that checks to see if the file is PDF and, if so, calls a hypothetical `OutputPDF()` function to output PDFs to `stdout`) to the beginning of your implementation of `OutputBinary()`.

```
void MyClientUser::OutputBinary( const char *data, int length )
{
    static unsigned char pdfFlag[] = { '%', 'P', 'D', 'F', '-' };
    if ( length >= 5 && memcmp( data, pdfFlag, sizeof( pdfFlag ) ) )
         OutputPDF( data, length );
    else
        ClientUser::OutputBinary( data, length );
}
```

ClientUser::OutputError(const char *)

Display a message as an error.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`const char *errBuf`	the error message
Returns	`void`	

Notes

The default implementation sends its argument to `stderr`. `OutputError()` is called by functions like `HandleError()`.

Example

Because the default implementation of `HandleError()` calls it, `OutputError()` is responsible for printing every error message in Helix server. For example:

```
p4 files //nonexistent/...
nonexistent - no such file(s).
```

In this case, the argument to `OutputError()` is the array containing the error message "nonexistent - no such file(s)."

To change the way error messages are displayed, create a subclass of `ClientUser` and define an alternate implementation of `OutputError()`.

For example, to print all error messages to `stdout` rather than `stderr`, and precede them with the phrase “!!ERROR!!”, implement `OutputError()` as follows:

```
void MyClientUser::OutputError( const char *errBuf )
{
    printf( "!!ERROR!! " );
    fwrite( errBuf, 1, strlen( errBuf ), stdout );
}
```

*ClientUser::OutputInfo(char, const char *)*

Output tabular data.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`char level`	the indentation "level" of the output
	`const char *data`	one line of output

Returns	`void`

Notes

`OutputInfo()` is called by the server during most Helix server commands; its most common use is to display listings of information about files. Any output not printed with `OutputInfo()` is typically printed with `OutputText()`. Running `p4 -s <command>` indicates whether any given line of output is "info" or "text".

In the default implementation of `OutputInfo()`, one "…" string is printed per "level". Values given as "levels" are either `0`, `1`, or `2`. The "data" passed is generally one line, without a line break; `OutputInfo()` adds the newline when it prints the output.

To capture information directly from Helix server commands for parsing or storing rather than output to `stdout`, it is usually necessary to use an alternate implementation of `OutputInfo()`.

2002.1 and newer servers do not call `OutputInfo()` to display information. Instead, they call `Message()`. The default implementation of `Message()` calls `OutputInfo()` if its argument represents information instead of an error; older code that uses `OutputInfo()` can be used with the newer API and newer servers, so long as the default implementation of `Message()` is retained.

Example

The `p4 filelog` command produces tabular output:

```
> p4 filelog final.c
//depot/final.c
... #3 change 703 edit on 2001/08/24 by testuser@shire (text) 'fixed'
... ... copy into //depot/new.c#4
... #2 change 698 edit on 2001/08/24 by testuser@shire (text) 'buggy'
... ... branch into //depot/middle.c#1
... #1 change 697 branch on 2001/08/24 by testuser@shire (text) 'test'
... ... branch from //depot/old.c#1,#3
```

Each line of output corresponds to one call to `OutputInfo()`. The first line of output has a level of '0', the line for each revision has a level of '1', and the integration record lines have levels of '2'. (The actual "data" text for these lines does not include the "…" strings.)

To alter the way in which "info" output from the server is handled, create a subclass of `ClientUser` and provide an alternate implementation of `OutputInfo()`.

For example, to capture output in a set of `StrBuf` variables rather than display it to `stdout`, your `ClientUser` subclass must contain three `StrBufs`, one for each level of info output, as follows:

```
void MyClientUser::OutputInfo( char level, const char *data )
{
    switch( level )
```

```
    {
        default:
        case '0':
            myInfo0.Append( data );
            myInfo0.Append( "\n" );
            break;
        case '1':
            myInfo1.Append( data );
            myInfo1.Append( "\n" );
            break;
        case '2':
            myInfo2.Append( data );
            myInfo2.Append( "\n" );
            break;
    }
}
```

*ClientUser::OutputStat(StrDict *)*

Process tagged output.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`StrDict *varList`	a `StrDict` containing the information returned by the command
Returns	`void`	

Notes

Normally, the only Helix server command that sends output through `OutputStat()` is `p4 fstat`, which always returns tagged output. Some other commands can be made to return tagged output by setting the "tag" protocol variable, in which case the output is in the form of a `StrDict` suitable for passing to `OutputStat()` for processing.

It is generally easier to deal with tagged output than it is to parse standard output. The default implementation of `OutputStat()` passes each variable/value pair in the `StrDict` to `OutputInfo()` as a line of text with a level of "1", with the exception of the "func" var, which it skips. Alternate implementations can use tagged output to extract the pieces of information desired from a given command.

See also "ClientUser::OutputStatPartial( )" on the next page

Example

Consider the following output from **p4 fstat**:

```
> p4 fstat file.c
... depotFile //depot/file.c
... clientFile c:\depot\file.c
... isMapped
... headAction integrate
... headType text
... headTime 998644337
... headRev 10
... headChange 681
... headModTime 998643970
... haveRev 10
```

The **StrDict** passed to **OutputStat()** consists of eight variable/value pairs, one for each line of output, plus a "func" entry, which is discarded by the default implementation of **OutputStat()**. Other commands can be made to return tagged output through **OutputStat()** by using the -Ztag global option at the command line.

To process tagged output differently, create a subclass of **ClientUser** with an alternate implementation of **OutputStat()**. The following simple example demonstrates how the “headRev” and “haveRev” variables resulting from an “fstat” command can be easily extracted and manipulated.

Other commands provide **StrDict**s with different variable/value pairs that can be processed in similar ways; use **p4 -Ztag *command*** to get an understanding for what sort of information to expect.

```
void MyClientUser::OutputStat( StrDict *varList )
{
    StrPtr *headrev;
    StrPtr *haverev;

    headrev = varList->GetVar( "headRev" );
    haverev = varList->GetVar( "haveRev" );

    printf( "By default, revision numbers are returned as strings:\n"
);
    printf( "  Head revision number: %s\n", headrev->Text() );
    printf( "  Have revision number: %s\n", haverev->Text() );
```

```
    printf( "but revision numbers can be converted to integers:\n" );
    printf( "  Head revision number: %d\n", headrev->Atoi() );
    printf( "  Have revision number: %d\n", haverev->Atoi() );
}
```

ClientUser::OutputStatPartial( )

Indicate whether `ClientUser::OutputStatPartial()` succeeded.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`StrDict *`	a `StrDict` dictionary of values
Returns	`int`	`0`, the default, means the same data will be processed through "ClientUser::OutputStat(StrDict *)" on page 113 `1` means that the application implementing this method handles the results.

Notes

If the client uses the default of a call to "ClientUser::OutputStat(StrDict *)" on page 113 and the changelist is very large, the number of items in tagged mode might cause an issue with memory. For example, in some cases the output of `p4 -ztag describe` can be large. An alternative is to use `OutputStatPartial()` to handle the changelist in chunk by chunk, that is, one file at a time.

ClientUser::OutputText(const char *, int)

Output textual data.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`const char *errBuf`	the block of text to be printed
	`int length`	the length of the data
Returns	`void`	

Notes

The most common usage of `OutputText()` is in running `p4 print` on a text file.

Example

```
> p4 print -q file.txt
This is a text file.
It is called "file.txt"
```

The arguments to `OutputText()` in the preceding example are the pointer to the first character in the file contents, and the length of the file in bytes.

To alter the way in which `OutputText()` handles text data, create a subclass of `ClientUser` and provide an alternate implementation of `OutputText()`.

For example, suppose that your `ClientUser` subclass contains a `StrBuf` called `myData`, and you want to store the data in this `StrBuf` rather than dump it to `stdout`.

```
void MyClientUser::OutputText( const char *data, int length )
{
    myData.Set( data, length );
}
```

ClientUser::ProgressIndicator()

Returns nonzero if progress is to be reported, otherwise returns 0.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	None	
Returns	`int`	returns non-zero if progress indicators are desired, 0 otherwise

Notes

After you have created a `ClientProgress` object with `CreateProgress()`, you must also implement `ProgressIndicator()` to return 0 or 1 depending on whether or not you want to report progress.

See also

`ClientUser::CreateProgress()` `ClientProgress::Description()` `ClientProgress::Done()` `ClientProgress::Total()` `ClientProgress::Update()`

Example

The typical implementation of `ProgressIndicator()` returns 1, and you call it when you wish to enable progress reporting:

```
MyUserProgress::ProgressIndicator()
{
    return 1;
}
```

*ClientUser::Prompt(const StrPtr &, StrBuf &, int, Error *)*

Prompt the user and get a response.

Virtual?	Yes	
Class	`ClientUser`	
Arguments	`const StrPtr &msg`	the message with which to prompt the user
	`StrBuf &rsp`	where to put the user's response
	`int noEcho`	specifies whether echo should be turned off at the console
	`Error *e`	an `Error` object
Returns	`void`	

Notes

`Prompt()` is used in the default implementation of `HandleError()` to prompt the user to correct the error. `Prompt()` is also used by the interactive resolve routine to prompt for options.

Example

Consider the following user interaction with `p4 resolve`:

```
> p4 resolve file.c
c:\depot\file.c - merging //depot/file.c#2,#10
```

```
Diff chunks: 0 yours + 1 theirs + 0 both + 0 conflicting
Accept(a) Edit(e) Diff(d) Merge (m) Skip(s) Help(?) [at]: at
```

In the above example, the "msg" argument to `Prompt()` is the “Accept…[at\]:” string. The response, "at", is placed into the "rsp" `StrBuf`, which is sent to the server and processed as "accept theirs".

To alter the behavior of `Prompt()`, create a subclass of `ClientUser` and provide an alternate implementation of `Prompt()`.

For example, suppose that you are writing a GUI application and want each option in the interactive resolve to appear in a dialog box. A function called `MyDialog()` to create a dialog box containing the text of its argument and a text field, and return a character array with the user’s response, would look like this:

```
void MyClientUser::Prompt( const StrPtr &msg, StrBuf &buf, \
                           int noEcho, Error *e )
{
    buf.Set( MyDialog( msg.Text() ) );
}
```

*ClientUser::RunCmd(const char *, const char *, [...], Error *)*

Call an external program.

Virtual?	No	
Class	`ClientUser` (static)	
Arguments	`const char *command`	the executable to be called
	`const char *arg1`	the first argument
	`const char *arg2`	the second argument
	`const char *arg3`	the third argument
	`const char *arg4`	the fourth argument
	`const char *pager`	a pager, if any
	`Error *e`	an `Error` object to hold system errors
Returns	`void`	

Notes

`RunCmd()` is called when the client needs to call an external program, such as a merge or diff utility. `RunCmd()` stores any resulting errors in the specified `Error` object.

Example

If you select "d" for "Diff" during an interactive resolve, and both `P4DIFF` and `P4PAGER` are set in your environment, `RunCmd()` is called with the following arguments:

Argument	Value
`command`	`P4DIFF`
`arg1`	local file name
`arg2`	temp file name (depot file)
`arg3`	null
`arg4`	null
`pager`	`P4PAGER`

The `P4DIFF` program is called with the two file names as arguments, and the output is piped through the `P4PAGER` program.

See the examples for `Diff()` and `Merge()` for code illustrating the use of `RunCmd()`.

Error methods

This section lists the available Error methods.

Error::Clear()

Remove any error messages from an `Error` object.

Virtual?	No
Class	`Error`
Arguments	None
Returns	`void`

Notes

`Clear()` can be used if you need to clear an `Error` after having handled it in a way that does not automatically clear it.

Example

The following code attempts to establish a connection to a nonexistent server, displays the error's severity, clears the error, and shows that the error has been cleared:

```
ClientApi client;
Error e;

client.SetPort( "bogus:12345" );
client.Init( &e );

printf( "Error severity after Init() is is %d.\n", e.GetSeverity() );
e.Clear();
printf( "Error severity after Clear() is %d.\n", e.GetSeverity() );
```

Executing the preceding code produces the following output:

```
Error severity after Init() is 4.
Error severity after Clear() is 0.
```

*Error::Dump(const char *)*

Display an `Error` struct for debugging.

Virtual?	No	
Class	`Error`	
Arguments	`const char * trace`	a string to appear next to the debugging output
Returns	`void`	

Notes

`Dump()` can be used to determine the exact nature of an `Error` that is being handled. Its primary use is in debugging, as the nature of the output is more geared towards informing the developer than helping an end user.

Example

The following code attempts to establish a connection to a nonexistent server, and dumps the resulting error:

```
ClientApi client;
Error e;

client.SetPort( "bogus:12345" );
client.Init( &e );
```

```
e.Dump( "example" );
```

Executing the preceding code produces the following output:

```
Error example 0012FF5C
   Severity 4 (error)
   Generic 38
   Count 3
      0: 1093012493 (sub 13 sys 3 gen 38 args 1 sev 4 code 3085)
      0: %host%: host unknown.
      1: 1093012492 (sub 12 sys 3 gen 38 args 1 sev 4 code 3084)
      1: TCP connect to %host% failed.
      2: 1076240385 (sub 1 sys 8 gen 38 args 0 sev 4 code 8193)
      2: Connect to server failed; check $P4PORT.
      host = bogus
      host = bogus:12345
```

*Error::Fmt(StrBuf *)*

Format the text of an error into a `StrBuf`.

Virtual?	No	
Class	`Error`	
Arguments	`StrBuf *buf`	a pointer to the `StrBuf` to contain the formatted error
Returns	`void`	

Notes

The result of `Fmt()` is suitable for displaying to an end user; this formatted text is what the command line client displays when an error occurs.

If an error has no severity (`E_EMPTY`), `Fmt()` returns with no change to the `StrBuf`.

If the error has severity of info (`E_INFO`), the `StrBuf` is formatted.

If the error has any higher severity, the `StrBuf` argument passed to `Fmt()` is cleared and then replaced with the formatted error.

Example

The following example code displays an error's text:

```
if ( e.Test() )
{
    StrBuf msg;
    e.Fmt( &msg );
    printf( "ERROR:\n%s", msg.Text() );
}
```

*Error::Fmt(StrBuf *, int)*

Format the text of an error into a `StrBuf`, after applying formatting.

Virtual?	No	
Class	`Error`	
Arguments	`StrBuf *buf`	a pointer to the `StrBuf` to contain the formatted error
	`int opts`	formatting options
Returns	`void`	

Notes

The result of `Fmt()` is suitable for displaying to an end user; this formatted text is what the command line client displays when an error occurs.

If an error has no severity (`E_EMPTY`), `Fmt()` returns with no change to the `StrBuf`.

If the error has severity of info (`E_INFO`), the `StrBuf` is formatted.

If the error has any higher severity, the `StrBuf` argument passed to `Fmt()` is cleared and then replaced with the formatted error.

The `opts` argument is a flag or combination of flags defined by the `ErrorFmtOpts` enum. The default is `EF_NEWLINE`, which puts a newline at the end of the buffer.

Formatting options are as follows:

Argument	Value	Meaning
`EF_PLAIN`	`0x00`	perform no additional formatting.
`EF_INDENT`	`0x01`	indent each line with a tab (`\t`)
`EF_NEWLINE`	`0x02`	default - terminate buffer with a newline (`\n`)
`EF_NOXLATE`	`0x04`	ignore `P4LANGUAGE` setting

Example

The following example code displays an error's text, indented with a tab.

```
if ( e.Test() )
{
    StrBuf msg;
    e.Fmt( &msg, EF_INDENT );
    printf( "ERROR:\n%s", msg.Text() );
}
```

Error::GetGeneric()

Returns generic error code of the most severe error.

Virtual?	No	
Class	`Error`	
Arguments	None	
Returns	`int`	the "generic" code of the most severe error

Notes

For more sophisticated handling, use a "switch" statement based on the error number to handle different errors in different ways.

The generic error codes are not documented at this time.

Example

The following example attempts to establish a connection to a nonexistent server, and displays the resulting generic error code.

```
ClientApi client;
Error e;

client.SetPort( "bogus:12345" );
client.Init( &e );

if ( e.Test() ) printf( "Init() failed, error code %d.\n",
e.GetGeneric() );
```

Executing the preceding code produces the following output:

```
Init() failed, error code 38.
```

Error::GetSeverity()

Returns severity of the most severe error.

Virtual?	No	
Class	`Error`	
Arguments	None	
Returns	`int`	the severity of the most severe error

Notes

The severity can take the following values:

Severity	Meaning
`E_EMPTY (0)`	no error
`E_INFO (1)`	information, not necessarily an error
`E_WARN (2)`	a minor error occurred
`E_FAILED (3)`	the command was used incorrectly
`E_FATAL (4)`	fatal error, the command can't be processed

Example

The following code attempts to establish a connection to a server, and beeps if the severity is a warning or worse:

```
ClientApi client;
Error e;

client.SetPort( "magic:1666" );
client.Init( &e );

if ( e.GetSeverity() > E_INFO ) printf( "Uh-oh!%c\n", 13 );
```

Error::IsFatal()

Tests whether there has been a fatal error.

Virtual?	No	
Class	`Error`	
Arguments	None	
Returns	`int`	nonzero if error is fatal

Notes

This function returns nonzero if `GetSeverity()` `== E_FATAL`.

Example

The following code attempts to establish a connection to a server, and beeps if the severity is fatal:

```
ClientApi client;
Error e;

client.SetPort( "magic:1666" );
client.Init( &e );

if ( e.IsFatal() ) printf( "Fatal error!%c\n", 13 );
```

Error::IsWarning()

Tests whether the error is a warning.

Virtual?	No	
Class	`Error`	
Arguments	None	
Returns	`int`	nonzero if the most severe error is a warning

Notes

This function returns nonzero if `GetSeverity()` `== E_WARN`.

Example

The following code attempts to establish a connection to a server, and beeps if the severity is a warning:

```
ClientApi client;
Error e;

client.SetPort( "magic:1666" );
client.Init( &e );

if ( e.IsWarning() ) printf( "Warning!%c\n", 13 );
```

*Error::Net(const char *, const char *)*

Add a network-related error to an `Error`.

Virtual?	No	
Class	`Error`	
Arguments	`const char *op`	the network operation that was attempted
	`const char *arg`	relevant information about that operation
Returns	`void`	

Notes

To use an `Error` object to track network-related errors, use `Net()`. Note that network communication with the Helix server and related errors are already handled by lower levels of the client API.

Example

The following example adds an error message, related to a failure to bind to a network interface, to an `Error` object.

```
e.Net( "bind", service.Text() );
```

Error::operator << (int)

Add data to the text of an error message.

Virtual?	No	
	Class	`Error`
Arguments	`int arg`	text to be added to this `Error`

Returns	`Error &`	a reference to the modified `Error`

Notes

The "<<" operator can be used to add text to an error as if the error is an output stream. This operator is typically used in the implementation of other `Error` methods.

Note that an `Error` consists of more than its text, it's more useful to use `Set()` to establish a base `Error` and then add text into that, rather than merely adding text to an empty `Error` object.

Example

The following example creates an `Error` using `Set()` and the `<<` operator.

```
e.Set( E_WARN, "Warning, number " ) << myErrNum;
```

*Error::operator << (char *)*

Add data to the text of an error message.

Virtual?	No	
Class	`Error`	
Arguments	`char *arg`	text to be added to this `Error`
Returns	`Error &`	a reference to the modified `Error`

Notes

The "<<" operator can be used to add text to an error as if the error is an output stream. This operator is typically used in the implementation of other `Error` methods.

Note that an `Error` consists of more than its text, it's more useful to use `Set()` to establish a base `Error` and then add text into that, rather than merely adding text to an empty `Error` object.

Example

The following example creates an `Error` using `Set()` and the `<<` operator.

```
e.Set( E_WARN, "Warning! " ) << "Something bad happened";
```

Error::operator << (const StrPtr &)

Add data to the text of an error message.

Virtual?	No	
Class	`Error`	
Arguments	`const StrPtr &arg`	text to be added to this `Error`
Returns	`Error &`	a reference to the modified `Error`

Notes

See `Error::operator << (int)` for details.

Error::operator = (Error &)

Copy an error.

Virtual?	No	
Class	`Error`	
Arguments	`Error & source`	the `Error` to be copied
Returns	`void`	

Notes

The "=" operator copies one `Error` into another.

Example

The following example sets `Error` `e1` to equal `e2`.

```
Error e1, e2;
e1 = e2;
```

*Error::Set(enum ErrorSeverity, const char *)*

Add an error message to an `Error`.

Virtual?	No
Class	`Error`
Arguments	`enum ErrorSeverity s`
	`const char *fmt`
Returns	`void`

Notes

An `Error` can hold multiple error messages; `Set()` adds the error message to the `Error`, rather than replacing the `Error's` previous contents.

An `ErrorSeverity` is an `int` from 0-4 as described in the documentation on `GetSeverity()`.

Example

The following example adds a fatal error to an `Error` object.

```
Error e;
e.Set( E_FATAL, "Fatal error!");
```

Error::Set(ErrorId &)

Add an error message to an `Error`.

Virtual?	No	
Class	`Error`	
Arguments	`ErrorId& id`	the severity and text of the error message
Returns	`void`	

Notes

See `Error::Set( enum ErrSeverity, const char * )` for details.

An `ErrorId` is a `struct` containing an `int` (`s`) and a `const char *` (`fmt`).

*Error::Sys(const char *, const char *)*

Add a system error to an `Error`.

Virtual?	No	
Class	`Error`	
Arguments	`const char *op`	the system call that was attempted
	`const char *arg`	relevant information about that call
Returns	`void`	

Notes

To use an `Error` object to track errors generated by system calls such as file operations, use `Sys()`.

Example

The following example adds an error message, related to a failure to rename a file, to an `Error` object.

```
e.Sys( "rename", targetFile->Name() );
```

Error::Test()

Test whether an `Error` is non-empty.

Virtual?	No	
Class	`Error`	
Arguments	None	
Returns	`int`	nonzero if the error is non-empty

Notes

`Test()` returns nonzero if `GetSeverity()` `!= E_EMPTY`.

Example

The following code attempts to establish a connection to a server, and beeps if an error occurs:

```
ClientApi client;
Error e;

client.SetPort( "magic:1666" );
client.Init( &e );

if ( e.Test() ) printf( "An error has occurred.%c\n", 13 );
```

ErrorLog methods

This section lists the available ErrorLog methods.

ErrorLog::Abort()

Abort with an error status if an error is detected.

Virtual?	No
Class	`ErrorLog`
Arguments	None
Returns	`void`

Notes

If the error is empty (severity is `E_EMPTY`), `Abort()` returns. Otherwise `Abort()` causes the program to exit with a status of -1.

Example

`Abort()` is typically called after `Init()` or `Run()` to abort the program with a non-zero status if there has been a connection problem. The code in `p4api.cc` is one example:

```
ClientApi client;
Error e;

client.Init( &e );
ErrorLog::Abort();
```

If any errors are generated during `ClientApi::Init()`, the `Error` object is non-empty, and `Abort()` reports the connection error before terminating the program.

ErrorLog::Report()

Print the text of an error to `stderr`.

Virtual?	No
Class	`ErrorLog`
Arguments	None
Returns	`void`

Notes

`Report()` functions similarly to `Error::Fmt()`, but displays the text on `stderr` rather than copying it into a `StrBuf`.

Example

The following example displays the contents of an error.

```
ClientApi client;
Error e;

client.Init( &e );
ErrorLog::Report();
```

*ErrorLog::SetLog(const char *)*

Redirects this `Error's` `Report()` output to a file.

Virtual?	No	
Class	`ErrorLog`	
Arguments	`const char *file`	the file to serve as an error log
Returns	`void`	

Notes

After `SetLog()` is called on a given `Error` object, `Report()` directs its output to the specified file rather than `stderr`. This setting applies only to the specified `Error` object.

Example

The following example redirects an `Error's` output to a log file, and then writes the `Error's` text to that log file.

```
ClientApi client;
Error e;

ErrorLog::SetLog( "C:\Perforce\errlog" );
client.Init( &e );
ErrorLog::Report();
```

ErrorLog::SetSyslog()

Redirects this `Error's` `Report()` output to `syslog` on UNIX only.

Virtual?	No
Class	`ErrorLog`

Arguments	None
Returns	`void`

Notes

This method is only valid on UNIX. After it is called, the output of `Report()` is redirected to `syslog`, similar to `SetLog()`.

Example

The following example redirects an `Error's` output to `syslog`, and then outputs the `Error's` text to `syslog`.

```
ClientApi client;
Error e;

ErrorLog::SetSyslog();
client.Init( &e );
ErrorLog::Report();
```

*ErrorLog::SetTag(const char *)*

Changes the standard tag used by this `Error's` `Report()` method.

Virtual?	No	
Class	`ErrorLog`	
Arguments	`const char *tag`	the text of the new tag
Returns	`void`	

Notes

The default tag is “Error”. `SetTag()` sets the new tag for the specified `Error` object only.

Example

The following example resets the tag on an `Error` to be “NewError”.

```
ClientApi client;
Error e;
```

```
client.Init( &e );
ErrorLog::SetTag( "NewError" );
```

ErrorLog::UnsetSyslog()

Stop writing errors to `syslog`.

Virtual?	No
Class	`ErrorLog`
Arguments	None
Returns	`void`

Notes

`UnsetSyslog()` reverses the effect of `SetSyslog()` by resetting the `Error` object to output to `stderr`.

Example

The following example prints an error message to `syslog` and then resets the `Error` back to using `stderr` for output.

```
ClientApi client;
Error e;

client.Init( &e );
ErrorLog::SetSyslog();
ErrorLog::Report();
ErrorLog::UnsetSyslog();
```

FileSys methods

This section lists the available FileSys methods.

*FileSys::Chmod(FilePerm, Error *)*

Modify the file mode bits of the file specified by the `path` protected `FileSys` member.

Virtual?	Yes	
Class	`FileSys`	
Arguments	`FilePerm perms`	permissions to change the file, either `FPM_RO` (read only) or `FPM_RW` (read/write)
	`Error *error`	returned error status
Returns	`void`	

Notes

This method is called to make a client file writable (`FPM_RW`) when it is opened for `edit`, or to change it to read-only (`FPM_RO`) after a `submit`.

A `FilePerm` is an `enum` taking one of the following values:

Argument	Value	Meaning
`FPM_RO`	`0x00`	leave file read-only.
`FPM_RW`	`0x01`	allow read and write operations

Example

To use `Chmod()` to create a configuration file and set its permissions to read-only:

```
FileSys *f = FileSys::Create( FST_ATEXT );
Error e;

f->Set( "c:\\configfile.txt" );
f->Chmod( FPM_RO, &e );
```

To reimplement `Chmod()` under UNIX:

```
void FileSysDemo::Chmod( FilePerm perms, Error *e )
{
    int bits = IsExec() ? PERM_0777 : PERM_0666;

    if ( perms == FPM_RO )
        bits &= ~PERM_0222;

    if ( chmod( Name(), bits & ~myumask ) < 0 )
        e->Sys( "chmod", Name() );
```

```
    if ( DEBUG )
        printf( "Debug (Chmod): %s\n", Name() );
}
```

FileSys::Close(Error *)

Close the file specified by the `path` protected `FileSys` member and release any OS resources associated with the open file.

Virtual?	Yes	
Class	`FileSys`	
Arguments	`Error *error`	returned error status
Returns	`void`	

Notes

The default implementation of `Close()` is called every time a file that is currently `Open()` is no longer required. Typically, the handle that was returned for the `Open()` call is used to free up the resource.

Your implementation must correctly report any system errors that may occur during the close.

Example

To use `Close()` to close an open file:

```
FileSys *f = FileSys::Create( FST_ATEXT );
Error e;

f->Set( "c:\\configfile.txt" );
f->Open( FOM_WRITE, &e );
f->Close( &e );
```

To reimplement `Close()` to report errors using `Error::Sys()` and provide debugging output:

```
void FileSysDemo::Close( Error *e )
{
    if ( close( fd ) == -1 )
        e->Sys( "close", Name() );
```

```
    if ( DEBUG )
        printf( "Debug (Close): %s\n", Name() );
}
```

FileSys::Create(FileSysType)

Create a new **FileSys** object.

Virtual?	Yes	
Class	**FileSys**	
Arguments	**FileSysType type**	file type
Returns	**FileSys ***	a pointer to the new **FileSys**.

Notes

A **FileSysType** is an **enum** taking one of the values defined in **filesys.h**. The most commonly used **FileSysType**s are as follows:

Argument	Value	Meaning
FST_TEXT	**0x0001**	file is text
FST_BINARY	**0x0002**	file is binary
FST_ATEXT	**0x0011**	file is text, open only for append

Example

To use **Create()** to create a **FileSys** object for a log file (text file, append-only):

```
FileSys *f = FileSys::Create( FST_ATEXT );
```

FileSys::Open(FileOpenMode, Error *)

Open the file name specified by the **path** protected **FileSys** member for reading or writing as specified by the argument **FileOpenMode**.

Virtual?	Yes
Class	**FileSys**

Arguments	`FileOpenMode mode`	Mode to open the file, either `FOM_READ` (open for read) or `FOM_WRITE` (open for write)
	`Error *error`	returned error status
Returns	`void`	

Notes

The default implementation of `Open()` is called every time there is a need to create or access a file on the client workspace.

Operating systems typically return a handle to the opened file, which is then used to allow future read/write calls to access the file.

Your implementation must correctly report any system errors that may occur during the open.

Example

To use `open()` to open a log file for writing:

```
FileSys *f = FileSys::Create( FST_ATEXT );
Error e;
StrBuf m;
m.Append( "example: text to append to a log file\r\n" );

f->Set( "C:\\logfile.txt" );
f->Open( FOM_WRITE, &e );
f->Write( m.Text(), m.Length(), &e );
f->Close( &e );
```

To reimplement `Open()` to report errors with `Error::Sys()`, provide debugging output, and use the `FileSysDemo` member “fd” to hold the file handle returned from the `open()` system call:

```
void FileSysDemo::Open( FileOpenMode mode, Error *e )
{
    this->mode = mode;

    int bits = ( mode == FOM_READ ) ? O_RDONLY
                                    : O_WRONLY|O_CREAT|O_APPEND;

    if ( ( fd = open( Name(), bits, PERM_0666 ) ) < 0 )
    {
        e->Sys( mode == FOM_READ ? "open for read" : "open for write",
```

```
            Name() );
    }

    if ( DEBUG )
    {
        printf( "Debug (Open): '%s' opened for '%s'\n", Name(),
            mode == FOM_READ ? "read" : "write" );
    }
}
```

*FileSys::Read(const char *, int, Error *)*

Attempt to read `len` bytes of data from the object referenced by the file handle (returned by the `Open()` method) to the buffer pointed to by `buf`. Upon successful completion, `Read()` returns the number of bytes actually read and placed in the buffer.

Virtual?	Yes	
Class	`FileSys`	
Arguments	`const char *buf`	pointer to buffer into which to read data
	`int len`	length of data to read
	`Error *error`	returned error status
Returns	`int`	number of bytes actually read

Notes

The default implementation of `Read()` is called every time there is a need to read data from the file referenced by the `Open()` call.

Your implementation must correctly report any system errors that may occur during I/O.

Example

To use `Read()` to read a line from a log file:

```
char line[80];
m.Set( msg );
FileSys *f = FileSys::Create( FST_ATEXT );
Error e;
```

```
f->Set( "C:\\logfile.txt" );
f->Open( FOM_READ, &e );
f->Read( line, 80, &e );
f->Close( &e );
```

To reimplement **Read()** to report errors with **Error::Sys()**, provide debugging output, and use the **FileSysDemo** member "fd" to hold the file handle returned from the **read()** system call:

```
int FileSysDemo::Read( char *buf, int len, Error *e )
{
    int bytes;

    if ( ( bytes = read( fd, buf, len ) ) < 0 )
        e->Sys( "read", Name() );

    if ( DEBUG )
    {
        printf( "debug (Read): %d bytes\n", bytes );
    }

    return( bytes );
}
```

*FileSys::Rename(FileSys *, Error *)*

Rename the file specified by the **path** protected **FileSys** member to the file specified by the target **FileSys** object.

Virtual?	Yes	
Class	**FileSys**	
Arguments	**FileSys *target**	name of target for rename
	Error *error	returned error status
Returns	**void**	

Notes

On some operating systems, an unlink might be required before calling **Rename()**.

Your implementation must correctly report any system errors that may occur during the rename.

Example

To use **Rename()** to rename **/usr/logs/log2** to **/usr/logs/log1**:

```
FileSys *f1 = FileSys::Create( FST_TEXT );
FileSys *f2 = FileSys::Create( FST_TEXT );
Error e;

f1->Set( "/usr/logs/log1" );
f2->Set( "/usr/logs/log2" );

f1->Rename( f2, &e );
```

To reimplement **Rename()** to report errors with **Error::Sys()** and provide debugging output:

```
void FileSysDemo::Rename( FileSys *target, Error *e )
{
    if ( rename( Name(), target->Name() ) < 0 )
        e->Sys( "rename", Name() );

    if ( DEBUG )
        printf( "Debug (Rename): %s to %s\n", Name(), target->Name()
);
}
```

*FileSys::Set(const StrPtr *)*

Initializes the protected **StrBuf** variable **path** to the supplied filename argument; this **path** is used by other **FileSys** member functions when reading and writing to a physical file location.

Virtual?	Yes	
Class	**FileSys**	
Arguments	**const StrPtr *name**	filename for this **FileSys** object
Returns	**void**	

Notes

After creating a **FileSys** object, call **Set()** to supply it with a **path**.

Example

To use `Set()` to set a filename:

```
FileSys *f = FileSys::Create( FST_BINARY );
f->Set( "/tmp/file.bin" );
```

To reimplement `Set()` to provide debugging output:

```
void FileSysDemo::Set( const StrPtr &name )
{
    // Set must initialize the protected variable "path"
    // with the filename argument "name".

    path.Set( name );

    if ( DEBUG )
        printf( "debug (Set): %s\n", path.Text() );
}
```

FileSys::Stat()

Obtain information about the file specified by the `path` protected `FileSys` member.

Virtual?	Yes	
Class	`FileSys`	
Arguments	None	
Returns	`int`	0 for failure, or status bits as defined below

The status bits have the following meanings:

Status	Meaning
`0`	failure
`FSF_EXISTS (0x01)`	file exists
`FSF_WRITEABLE (0x02)`	file is user-writable
`FSF_DIRECTORY (0x04)`	file is a directory
`FSF_SYMLINK (0x08)`	file is symlink

Status	Meaning
`FSF_SPECIAL (0x10)`	file is a special file (in the UNIX sense)
`FSF_EXECUTABLE (0x20)`	file is executable
`FSF_EMPTY (0x40)`	file is empty
`FSF_HIDDEN (0x80)`	file is invisible (hidden)

Notes

The default implementation of `Stat()` is called to obtain file status every time a file is opened for read.

Example

To use `Stat()` to verify the existence of `/usr/bin/p4`:

```
FileSys *f = FileSys::Create( FST_BINARY );
f->Set( "/usr/bin/p4" );
int state = f->Stat();

if ( state & FSF_EXISTS )
    printf( "File found\n" );
```

To reimplement `Stat()` to provide debugging output:

```
int FileSysDemo::Stat()
{
    int flags = 0;
    struct stat st;

    if ( DEBUG )
        printf( "Debug (Stat): %s\n", Name() );

    if ( stat( Name(), &st ) < 0 )
        return( flags );

    // Set internal flags

    flags |= FSF_EXISTS;
```

```
    if ( st.st_mode & S_IWUSR ) flags |= FSF_WRITEABLE;
    if ( st.st_mode & S_IXUSR ) flags |= FSF_EXECUTABLE;
    if ( S_ISDIR( st.st_mode ) ) flags |= FSF_DIRECTORY;
    if ( !S_ISREG( st.st_mode ) ) flags |= FSF_SPECIAL;
    if ( !st.st_size ) flags |= FSF_EMPTY;
    return flags;
}
```

FileSys::StatModTime()

Return the last modified time of the file specified by the **`path`** protected **`FileSys`** member.

Virtual?	Yes	
Class	**`FileSys`**	
Arguments	None	
Returns	**`int`**	0 for failure, or last modified time in seconds since 00:00:00, January 1, 1970, GMT.

Notes

The default implementation of **`StatModTime()`** is called every time a client file is submitted or synced.

Example

To use **`StatModTime()`** to obtain the modification time on a log file:

```
FileSys *f = FileSys::Create( FST_ATEXT );
f->Set( "/usr/logs/logfile.txt" );
int time = f->StatModTime();

if ( time )
    printf( "%d", time );
```

To reimplement **`StatModTime()`** to provide debugging output:

```
int FileSysDemo::StatModTime()
{
    struct stat st;
```

```
    if ( stat( Name(), &st ) < 0 )
        return( 0 );

    if ( DEBUG )
        printf( "Debug (StatModTime): %s\n", Name() );

    return (int)( st.st_mtime );
}
```

FileSys::Truncate()

Truncate the file specified by the `path` protected `FileSys` member to zero length.

Virtual?	Yes
Class	`FileSys`
Arguments	None
Returns	`void`

Notes

The default implementation of `Truncate()` is only called by the Helix server.

FileSys::Unlink(Error *)

Remove the file specified by the `path` protected `FileSys` member from the filesystem.

Virtual?	Yes	
Class	`FileSys`	
Arguments	`Error *error`	returned error status
Returns	`void`	

Notes

The default implementation of `Unlink()` is always called if the file created is temporary.

Your implementation must correctly report any system errors that may occur during removal.

Example

To use `Unlink()` to delete an old log file:

```
FileSys *f = FileSys::Create( FST_TEXT );
Error e;

f->Set( "/usr/logs/oldlog" );
f->Unlink( &e );
```

To reimplement `Unlink()` to report errors with `Error::Sys()` and provide debugging output:

```
void FileSysDemo::Unlink( Error *e )
{
    if ( unlink( Name() ) < 0 )
        e->Sys( "unlink", Name() );

    if ( DEBUG )
        printf( "Debug (Unlink): %s\n", Name() );
}
```

*FileSys::Write(const char *, int, Error *)*

Attempt to write “len” bytes of data to the object referenced by the file handle (returned by the `Open()` method) from the buffer pointed to by “buf”.

Virtual?	Yes	
Class	`FileSys`	
Arguments	`const char *buf`	pointer to buffer containing data to be written
	`int len`	length of data to write
	`Error *error`	returned error status
Returns	`void`	

Notes

The default implementation of `Write()` is called every time there is a need to write data to the file created by the `Open()` call.

Your implementation must correctly report any system errors that may occur during I/O.

Example

To use `Write()` to write an error to a log file:

```
StrBuf m;
m.Set( "Unknown user\r\n" );
FileSys *f = FileSys::Create( FST_ATEXT );
Error e;

f->Set( "C:\\logfile.txt" );
f->Open( FOM_WRITE, &e );
f->Write( m.Text(), m.Length(), &e );
f->Close( &e );
```

To reimplement `Write()` to report errors with `Error::Sys()` and provide debugging output:

```
void FileSysDemo::Write( const char *buf, int len, Error *e )
{
    int bytes;

    if ( ( bytes = write( fd, buf, len ) ) < 0 )
        e->Sys( "write", Name() );

    if ( DEBUG )
        printf( "debug (Write): %d bytes\n", bytes );
}
```

Ignore methods

This section lists the available Ignore methods.

Ignore::Reject(const StrPtr &, const StrPtr &)

Tests whether the provided path will be rejected when it is opened for add because it matches an entry in the provided ignore file.

Virtual?	No	
Class	`Ignore`	
Arguments	`const StrPtr &path`	the path to check

	`const StrPtr &ignoreFile`	the full path to the ignore file
Returns	`int`	nonzero if path is ignored

Notes

Calling `Reject()` provides a preview of what will happen when files are opened for add.

If the ignore file does not exist, or is not readable, no files are rejected.

Example

The following example demonstrates the usage of `Reject()`.

```
# include "clientapi.h"
# include "ignore.h"

int main()
{
    ClientApi client;
    StrBuf clientPath;

    client.SetIgnoreFile( ".p4ignore" );
    clientPath = "ignore.txt";
    if ( client->GetIgnore()->Reject( *clientPath,
                                      client->GetIgnoreFile() ) )
    {
        printf( "%s is to be ignored.\n", clientPath.Text() );
    }
}
```

Ignore::RejectCheck(const StrPtr &)

Tests whether the provided path will be rejected when it is opened for add because it matches an ignore file entry.

Virtual?	No	
Class	`Ignore`	
Arguments	`cont StrPtr &path`	the path to check
Returns	`int`	nonzero if path is ignored

Notes

Calling `RejectCheck()`. provides a preview of what will happen the file is opened for add.

Use `RejectCheck()` when you have to test multiple paths that may be rejected. First call `Reject()` to parse the ignore file, and then call `RejectCheck()` for each additional path that needs to be checked.

Example

The following example demonstrates the usage of `RejectCheck()`.

```
# include "clientapi.h"
# include "ignore.h"

int main()
{
    ClientApi client;
    StrBuf clientPath;

    client.SetIgnoreFile( ".p4ignore" );
    clientPath = "ignore.txt";
    if ( client->GetIgnore()->Reject( *clientPath,
                                      client->GetIgnoreFile() ) )
    {
        printf( "%s is to be ignored.\n", clientPath.Text() );
    }

    clientPath = "ignore2.txt";
    if ( client->GetIgnore()->Reject( *clientPath,
                                      client->GetIgnoreFile() ) )
    {
        printf( "%s is to be ignored.\n", clientPath.Text() );
    }
}
```

KeepAlive methods

This section lists the available KeepAlive methods.

KeepAlive::IsAlive()

The only method of the `KeepAlive` class, `IsAlive()` is used in applications to request that the current command be terminated by disconnecting.

Virtual?	Yes	
Class	`KeepAlive`	
Arguments	None	
Returns	`int`	0 to terminate connection, 1 to continue processing

Notes

Use `ClientApi::SetBreak()` to establish a callback to be called every 0.5 seconds during command execution.

See also

`ClientApi::SetBreak()`

Example

The following example implements a custom `IsAlive()` that can be called three times before returning 0 and terminating the connection. If the call to run the `changes` command takes less than 1.5 seconds to complete on the server side, the program outputs the list of changes. If the call to run the `changes` command takes more than 1.5 seconds, the connection is interrupted.

```
#include <clientapi.h>

// subclass KeepAlive to implement a customized IsAlive function.
class MyKeepAlive : public KeepAlive
{
    public:
    int  IsAlive();
};

// Set up the interrupt callback. After being called 3 times,
// interrupt 3 times, interrupt the current server operation.
int   MyKeepAlive::IsAlive()
{
    static int counter = 0;
```

```
    if ( ++counter > 3 )
    {
        counter = 0;
        return( 0 );
    }
    return( 1 );
}

// Now test the callback
ClientUser ui;
ClientApi client;
MyKeepAlive cb;
Error e;

client.Init( &e );
client.SetBreak( &cb );   // SetBreak must happen after the Init
client.Run( "changes", &ui );
client.Final( &e );
```

MapApi methods

This section lists the available MapApi methods.

MapApi::Clear()

Empties a mapping.

Virtual?	No
Class	`MapApi`
Arguments	None
Returns	`void`

Notes

After this method has been called on a `MapApi` object, the object is indistinguishable from a freshly-constructed object.

MapApi::Count()

Returns the number of entries currently in the mapping.

Virtual?	No	
Class	`MapApi`	
Arguments	None	
Returns	`int`	The number of entries currently in the mapping

Notes

The number returned by `Count()` may be different from the number of times that `Insert()` has been called. This is because `MapApi` automatically disambiguates itself, adding new exclusions to eliminate ambiguity between partially overlapping entries and removing entries that are redundant.

Example

The following example demonstrates `Count()`, `GetType()`, `GetLeft()`, and `GetRight()` being used to iterate over a `MapApi` that contains four entries after two calls to `Insert()`.

This code produces the following output:

```
//depot/... //client/...
-//depot/d2/... //client/d2/...
-//depot/d1/... //client/d1/...
//depot/d1/... //client/d2/...
```

```
MapApi clientmap;

clientmap.Insert( StrRef( "//depot/..." ), StrRef( "//client/..." ) );
clientmap.Insert( StrRef( "//depot/d1/..." ), StrRef(
"//client/d2/..." ) );

char c = ' ';
for ( int i = 0; i < clientmap.Count(); i++ )
{
    switch( clientmap.GetType( i ) )
    {
        case MapInclude:
            c = ' '; break;
```

```
        case MapExclude:
            c = '-'; break;
        case MapOverlay:
            c = '+'; break;
    }

    printf( "%c%s %s\n", c,
            clientmap.GetLeft( i )->Text(),
            clientmap.GetRight( i )->Text() );
}
```

MapApi::GetLeft(int)

Returns the left side of the specified view entry.

Virtual?	No	
Class	`MapApi`	
Arguments	`int i`	the index of the desired entry
Returns	`const StrPtr *`	a string representing the left side of the entry

Notes

The index should be between 0 and one less than the number of mapping entries.

See also

`MapApi::Count()`

Example

See the example for `MapApi::Count()`.

MapApi::GetRight(int)

Returns the right side of the specified view entry.

Virtual?	No
Class	`MapApi`

Arguments	`int i`	the index of the desired entry
Returns	`const StrPtr *`	a string representing the right side of the entry

Notes

The index should be between 0 and one less than the number of mapping entries.

See also

`MapApi::Count()`

Example

See the example for `MapApi::Count()`.

MapApi::GetType(int)

Returns the type of the specified view entry.

Virtual?	No	
Class	`MapApi`	
Arguments	`int i`	the index of the desired entry
Returns	`MapType`	the entry type

Notes

The entry type is one of `MapInclude`, `MapExclude`, and `MapOverlay`.

`MapExclude` entries negate earlier `MapInclude` and `MapOverlay` entries that map the same paths, and `MapOverlay` entries are not disambiguated if they overlap with earlier `MapInclude` entries.

In human-readable Helix server view specifications, `MapExclude` lines are indicated with a `-` character, and `MapOverlay` lines are indicated with a `+` character.

See also

`MapApi::Count()`

Example

See the example for `MapApi::Count()`.

MapApi::Insert(const StrPtr &, MapType)

Adds a new entry to the mapping.

Virtual?	No	
Class	`MapApi`	**Arguments**
`StrPtr & lr`	the path to which the entry applies	
`MapType t`	the mapping type (by default, `MapInclude`)	**Returns**

Notes

This `Insert()` overload is a convenience function that adds an entry with identical left and right sides. It is meant to represent mappings whose sole purpose is to include and exclude files, such as protection tables and label views.

Example

The following example demonstrates the construction and use of a protection table mapping.

```
MapApi protect;
protect.Insert( StrRef( "//..." ) );
protect.Insert( StrRef( "//private/..." ), MapExclude );

StrBuf to;
StrBuf file1( "//depot/file.txt" );
StrBuf file2( "//private/file.txt" );

printf( "%s - access %d\n", file1.Text(), protect.Translate( file1, to
) );

printf( "%s - access %d\n", file2.Text(), protect.Translate( file2, to
) );
```

This produces the following output:

```
//depot/file.txt - access 1
//private/file.txt - access 0
```

MapApi::Insert(const StrPtr &, const StrPtr &, MapType)

Adds a new entry to the mapping.

Virtual?	No	
Class	`MapApi`	
Arguments	`StrPtr & l`	the left side of the entry
	`StrPtr & r`	the right side of the entry
	`MapType t`	the mapping type (by default, `MapInclude`)
Returns	`void`	

Notes

`Insert()` adds one new entry to a mapping at the "bottom" (highest precedence) position in the map. The `MapType` parameter indicates whether the entry is a standard inclusion (the default), an exclusion, or an overlay mapping (only useful when modeling a client view).

Example

The following example demonstrates the construction and use of a branch view mapping.

```
MapApi branch;
branch.Insert( StrRef( "//depot/main/..." ), StrRef(
"//depot/rel1/..." ) );

StrBuf source( "//depot/main/file.c" );
StrBuf target;

branch.Translate( source, target );
printf( "%s -> %s\n", source.Text(), target.Text() );
```

This produces the following output:

```
//depot/main/file.c -> //depot/rel1/file.c
```

*MapApi::Join(MapApi *, MapApi *) [static]*

Joins two `MapApi`s together to produce a combined mapping.

Virtual?	No	
Class	`MapApi`	
Arguments	`MapApi *left`	the first mapping

	`MapApi *right`	the second mapping
Returns	`MapApi *`	a new MapApi representing the joined maps

Notes

This overload of `Join()` links the right side of the first mapping to the left side of the second mapping, as if the two mappings were laid out left to right and glued together in the middle. The resulting `MapApi`'s left side corresponds to the first mapping's left side, and its right side corresponds to the second mapping's right side.

If the right side of the first mapping does not have anything in common with the left side of the second mapping, the resulting map is empty.

The other `Join()` overload allows more control over which side of each mapping is joined to the other, and the direction of the resulting mapping.

This function allocates a new `MapApi` object on the heap; the caller is responsible for deleting it.

Example

The following example demonstrates a join between a branch view and a client view.

```
MapApi branchmap;
branchmap.Insert( StrRef( "//depot/main/..." ), StrRef(
"//depot/rel1/...") );

MapApi clientmap;
clientmap.Insert( StrRef( "//depot/..." ), StrRef(
"//client/depot/..." ) );

MapApi *branch_to_client = MapApi::Join( &branchmap, &clientmap );

StrBuf source( "//depot/main/file.c" );
StrBuf target;

branch_to_client->Translate( source, target );
printf( "%s -> %s\n", source.Text(), target.Text() );
delete branch_to_client;
```

This produces the following output:

```
//depot/main/file.c -> //client/depot/rel1/file.c
```

MapApi::Join(MapApi *, MapDir, MapApi *, MapDir) [static]

Joins two `MapApi`s together to produce a combined mapping.

Virtual?	No	
Class	`MapApi`	
Arguments	`MapApi *m1`	the first mapping
	`MapDir d1`	the orientation of the first mapping
	`MapApi *m2`	the second mapping
	`MapDir d2`	the orientation of the second mapping
Returns	`MapApi *`	a new MapApi representing the joined maps

Notes

This overload of `Join()` works exactly like the simpler two-argument overload, but allows the caller to reverse either or both mappings before they are joined together. Specifying `MapLeftRight` as the direction for both mappings will produce the same result as the two-argument `Join()`.

If the two mappings do not have anything in common at the join point, the result is an empty mapping.

This function allocates a new `MapApi` object on the heap; the caller is responsible for deleting it.

Example

The following example demonstrates a join between a branch view and a client view, with both mappings reversed so that the client path is on the left side of the result and the branch source is on the right side.

```
MapApi branchmap;
branchmap.Insert( StrRef( "//depot/main/..." ), StrRef(
"//depot/rel1/...") );

MapApi clientmap;
clientmap.Insert( StrRef( "//depot/..." ), StrRef(
"//client/depot/..." ) );

MapApi *client_to_branch = MapApi::Join
    ( &clientmap, MapRightLeft, &branchmap, MapRightLeft );

StrBuf clientFile( "//client/depot/rel1/file.c" );
```

```
StrBuf branchFile;

client_to_branch->Translate( clientFile, branchFile );
printf( "%s -> %s\n", clientFile.Text(), branchFile.Text() );
delete client_to_branch;
```

Executing the preceding code produces the following output:

```
//client/depot/rel1/file.c -> //depot/main/file.c
```

MapApi::Translate(const StrPtr &, StrBuf&, MapDir)

Translates a file path from one side of a mapping to the other.

Virtual?	No	
Class	`MapApi`	
Arguments	`const StrPtr & from`	the input path
	`StrBuf & to`	the output path
	`MapDir d`	the direction in which to translate (by default, `MapLeftRight`)
Returns	`bool`	whether or not the translation succeeded

Notes

The `Translate()` function is used to determine the effect of the mapping on any particular file. In the case of a two-sided mapping (such as a client view), it indicates where any given depot file maps in the client, or vice versa. In the case of a one-sided mapping (such as a protection table), it simply indicates whether a particular file is mapped at all.

If the specified `MapDir` is `MapLeftRight`, the input path is translated from the left side of the mapping to the right side of the mapping. If the `MapDir` is `MapRightLeft`, the mapping is effectively inverted, so that the input path is translated from the right side to the left.

If the input path does not match anything in the left side of the mapping (or the right side in the `MapRightLeft` case), the translation fails just as if the input path had been excluded from the mapping.

`Translate()` is designed to map single files. To model the effect of passing a broader path through a mapping, create a new one-sided mapping that represents that path and `Join()` it with the other mapping.

Examples

See the examples for `Insert()` and `Join()`.

Options methods

This section lists the available Options methods.

Options::GetValue(char opt, int subopt)

Returns the value of a flag previously stored by `Options::Parse()`.

Virtual?	No	
Class	`Options`	
Arguments	`char opt`	The flag to check
	`int subopt`	Return the argument associated with the `subopt`-th occurrence of the `opt` flag on the command line.
Returns	`StrPtr *`	The value of the flag. This is “true” for flags which, when provided, do not take a value, and `NULL` if the flag is not provided

Notes

You must call `Options::Parse()` before calling `GetValue()`.

If a flag does not occur on the command line, `GetValue()` returns `NULL`.

If a flag is provided without a value, `GetValue()` returns “true”.

If a flag appears only once on a command line, extract the value of its arguments by calling `GetValue()` with a `subopt` of zero, or use the `[]` operator.

If a flag occurs more than once on a command line, extract the value supplied with each occurrence by calling `Options::GetValue()` once for each occurrence, using different `subopt` values.

See also

`Options::Parse()` `Options::operator[]`

Example

Executing the following code produces the following output:

```
$ getvalue -h -c1 -c2 -d3
```

```
opts.GetValue( h, 0 ) value is true
opts.GetValue( c, 0 ) value is 1
opts.GetValue( c, 1 ) value is 2
opts.GetValue( d, 0 ) value is 3
```

```
#include <stdhdrs.h>
#include <strbuf.h>
#include <error.h>
#include <options.h>

int main( int argc, char **argv )
{
    // Parse options.
    Error *e = new Error();
    ErrorId usage = { E_FAILED, "Usage: getvalue -h for usage." };

    Options opts;

    // strip out the program name before parsing
    argc--;
    argv;

    char *ParseOpts = "ha:b:c:d:e:f:";
    opts.Parse( argc, argv, ParseOpts, OPT_ANY, usage, e );

    if ( e-&gt;Test() )
    {
        StrBuf msg;
        e-&gt;Fmt( &amp;msg );   // See Error::Fmt()
        printf( "ERROR:\n%s", msg.Text() );
        return 1;
    }

    char *iParseOpts = ParseOpts;
    int isubopt;
    StrPtr *s;
```

```
    // Print values for options.
    while( *iParseOpts != '\0' )
    {
        if ( *iParseOpts != ':' )
        {
            isubopt = 0;
            while( s = opts.GetValue( *iParseOpts, isubopt ) )
            {
                printf( "opts.GetValue( %c, %d ) value is %s\n",
                        *iParseOpts, isubopt, s-&gt;Text() );
                isubopt;
            }
        }
        iParseOpts++;
    }
    return 0;
}
```

Options::operator[](char opt)

Returns the value of a flag previously stored by `Options::Parse()`.

Virtual?	No	
Class	`Options`	
Arguments	`char opt`	The flag to check

Notes

You must call `Options::Parse()` before using the `[]` operator.

If a flag does not occur on the command line, the `[]` operator returns `NULL`.

If a flag is provided without a value, the `[]` operator returns “true”.

If a flag appears once on a command line, the `[]` operator returns its argument. This is equivalent to calling `Options::GetValue()` with a `subopt` of zero.

The `[]` operator is sufficient for extracting the value of any flag which does not have more than one value associated with it. If a flag appears more than once on the same command line, you must use `Options::GetValue()`, specifying a different `subopt` value for each appearance.

See also

`Options::Parse()` `Options::GetValue()`

Example

The following code parses some of the standard Helix server global options and stores them in a `ClientApi` object.

If the -h option is supplied, the program also displays a brief message.

```
#include <iostream>
#include <clientapi.h>
#include <error.h>
#include <errornum.h>
#include <msgclient.h>
#include <options.h>

int main( int argc, char **argv )
{
    Error *e = new Error();
    ErrorId usage = { E_FAILED, "Usage: myapp -h for usage." };

    // Bypass argv[0] before parsing
    argc--;
    argv++;

    Options opts;
    opts.Parse( argc, argv, "hc:H:d:u:p:P:", OPT_ANY, usage, e );

    if ( e->Test() )
    {
        StrBuf msg;
        e->Fmt( &msg );   // See Error::Fmt()
        printf( "Error: %s", msg.Text() );
        return 1;
    }

    ClientApi client;
```

```
    StrPtr *s;

    // Get command line overrides of client, host, cwd, user, port,
pass
    if ( s = opts[ 'h' ] ) printf ( "User asked for help\n" );
    if ( s = opts[ 'c' ] ) client.SetClient ( s );
    if ( s = opts[ 'H' ] ) client.SetHost ( s );
    if ( s = opts[ 'd' ] ) client.SetCwd ( s );
    if ( s = opts[ 'u' ] ) client.SetUser ( s );
    if ( s = opts[ 'p' ] ) client.SetPort ( s );
    if ( s = opts[ 'P' ] ) client.SetPassword ( s );

    // Perform desired operation(s) with your ClientApi here
    return 0;
}
```

*Options::Parse(int &, char ** &, const char *, int, const ErrorId &, Error *)*

Manipulate `argc` and `argv` to extract command line arguments and associated values.

Virtual?	No	
Class	`Options`	
Arguments	`int &argc`	Number of arguments
	`char ** &argv`	An array of arguments to parse
	`const char *opts`	The list of valid options to extract
	`int flag`	A flag indicating how many arguments are expected to remain when parsing is complete
	`const ErrorId &usage`	An error message containing usage tips
	`Error *e`	The `Error` object to collect any errors encountered
Returns	`void`	

Notes

You must bypass `argv[0\]` (that is, the name of the calling program) before calling `Options::Parse()`. This is most easily done by decrementing `argc` and incrementing `argv`.

An argument by be of the form -*avalue* or -*avalue*. Although an argument of the form -*avalue* is passed as two entries in `argv`, the `Options::Parse()` method parses it as one logical argument.

As arguments are scanned from the caller's `argv`, the caller's `argc` and `argv` are modified to reflect the arguments scanned. Scanning stops when the next argument either:

- does not begin with a `-`, or
- is a `-` only, or
- is not in the array of expected options.

Once scanning has stopped, `argc` and `argv` are returned "as-is"; that is, they are returned as they were when scanning stopped. There is no "shuffling" of arguments.

The `opts` argument is a format string indicating which options are to be scanned, and whether these options are to have associated values supplied by the user. Flags with associated values must be followed by a colon (":") or a period (".") in the format string. Using a colon allows arguments to be specified in the form -*avalue* or -*avalue*; using a period allows only the -*avalue* form.

If, based on the expectation set in the format string, the actual option string in `argv` does not provide a value where one is expected, an error is generated.

For instance, the `p4` Command Line Client's -V and -? flags are expected to be supplied without values, but the -p flag is expected to be accompanied with a setting for `P4PORT`. This is the format string used by the `p4` Command Line Client:

```
"?c:C:d:GRhH:p:P:l:L:su:v:Vx:z:Z:"
```

Characters followed by colons (`c`, `C`, and so on) are command line flags that take values; all characters not followed by colons (`?`, `G`, `R`, `h`, `s`, and `V`) represent command line flags that require no values.

There is a limit of 20 options per command line, as defined in `options.h` by the constant `N_OPTS`.

The `flag` argument should be one of the following values (defined in `options.h`):

Argument	Value	Meaning
`OPT_ONE`	`0x01`	Exactly one argument is expected to remain after parsing
`OPT_TWO`	`0x02`	Exactly two arguments are expected to remain after parsing
`OPT_THREE`	`0x04`	Exactly three arguments are expected to remain after parsing
`OPT_MORE`	`0x08`	More than two arguments (three or more) are to remain after parsing
`OPT_NONE`	`0x10`	Require that zero arguments remain after parsing; if arguments remain after parsing, set an error.

Argument	Value	Meaning
`OPT_MAKEONE`	`0x20`	If no arguments remain after parsing, create one that points to `NULL`.
`OPT_OPT`	`0x11`	`NONE`, or `ONE`.
`OPT_ANY`	`0x1F`	`ONE`, `TWO`, `THREE`, `MORE`, or `NONE`.
`OPT_DEFAULT`	`0x2F`	`ONE`, `TWO`, `THREE`, `MORE`, or `MAKEONE`.
`OPT_SOME`	`0x0F`	`ONE`, `TWO`, `THREE`, or `MORE`.

See also

`Options::GetValue()` `Options::operator[]()`

Example

The following code and examples illustrate how `Options::Parse()` works.

```
#include <stdhdrs.h>
#include <strbuf.h>
#include <error.h>
#include <options.h>

int main( int argc, char **argv )
{
    // Parse options.
    Error *e = new Error();
    ErrorId usage = { E_FAILED, "Usage: parse optionstring flag args"
};

    Options opts;

    // strip out the program name before parsing
    argc--;
    argv;

    // next argument is options to be parsed
    char *ParseOpts = argv[ 0 ];
```

```
    argc--;
    argv;

    // next argument is number of arguments remaining after parse
    int flag = strtol( argv[ 0 ], NULL, 0 );
    argc--;
    argv;

    // Echo pre-parse values
    int iargv;
    printf( "Prior to Options::Parse call:\n" );
    printf( "  ParseOpts is %s\n", ParseOpts );
    printf( "  flag is 0x%2.2X\n", flag );
    printf( "  argc is %d\n", argc );
    for ( iargv = 0; iargv &lt; argc; iargv )
    {
        printf( "  argv[ %d ] is %s\n", iargv, argv[ iargv ] );
    }
    printf( "\n" );

    opts.Parse( argc, argv, ParseOpts, flag, usage, e );
    if ( e->Test() )
    {
        // See example for Error::Fmt()
        StrBuf msg;
        e->Fmt( &msg );
        printf( "ERROR:\n%s\n", msg.Text() );
    }

    char *iParseOpts = ParseOpts;
    int isubopt;
    StrPtr *s;

    // Print values for options.
    while( *iParseOpts != '\0' )
```

```
    {
        if ( *iParseOpts != ':' )
        {
            isubopt = 0;
            while( s = opts.GetValue( *iParseOpts, isubopt ) )
            {
                printf( "opts.GetValue( %c, %d ) value is %s\n",
                        *iParseOpts, isubopt, s->Text() );

                isubopt;
            }
        }

        iParseOpts;
    }

    // Echo post-parse values
    printf( "\n" );
    printf( "After Options::Parse call:\n" );
    printf( "  argc is %d\n", argc );
    for ( iargv = 0; iargv < argc; iargv++ )
    {
        printf( "  argv[ %d ] is %s\n", iargv, argv[ iargv ] );
    }

    return 0;
}
```

Invoke **`parsedemo`** with a format string, a flag (as defined in **`options.h`**) to specify the number of options expected, and a series of arguments.

For instance, to allow arguments -a, -b and -c, where -a and -b take values, but -c does not take a value, and to use a **`flag`** of **`OPT_NONE`** (**`0x10`**) to require that no options remain unparsed after the call to **`Options::Parse()`**, invoke **`parsedemo`** as follows.

```
$ parsedemo a:b:c 0x10 -a vala -b valb -c
```

Arguments of the form **`-c one`** are passed as two entries in **`argv`**, but parsed as one logical argument:

```
$ parsedemo ha:b:c:d:e: 0x10 -cone
Prior to Options::Parse call:
  ParseOpts is ha:b:c:d:e:
  flag is 0x10
  argc is 1
  argv[ 0 ] is -cone

opts.GetValue( c, 0 ) value is one

After Options::Parse call:
  argc is 0

$ parsedemo ha:b:c:d:e: 0x10 -c one
Prior to Options::Parse call:
  ParseOpts is ha:b:c:d:e:
  flag is 0x10
  argc is 2
  argv[ 0 ] is -c
  argv[ 1 ] is one

opts.GetValue( c, 0 ) value is one

After Options::Parse call:
  argc is 0
```

Use of a period in the options string disallows the **`-c one`** form for the **`c`** option:

```
$ parsedemo ha:b:c.d:e: 0x10 -c one
Prior to Options::Parse call:
  ParseOpts is ha:b:c.d:e:
  flag is 0x10
  argc is 2
  argv[ 0 ] is -c
  argv[ 1 ] is one

ERROR:
Usage: parse optionstring flag args
```

```
Unexpected arguments.

opts.GetValue( c, 0 ) value is

After Options::Parse call:
  argc is 1
  argv[ 0 ] is one
```

Arguments not in the format string are permitted or rejected with the use of different flag values; **OPT_NONE** (**0x10**) requires that no arguments remain after the call to **Options::Parse()**, while **OPT_ONE** (**0x01**) requires that one argument remain.

```
$ parsedemo ha:b:c:d:e: 0x10 -c one two
Prior to Options::Parse call:
  ParseOpts is ha:b:c:d:e:
  flag is 0x10
  argc is 3
  argv[ 0 ] is -c
  argv[ 1 ] is one
  argv[ 2 ] is two

ERROR:
Usage: parse optionstring flag args
Unexpected arguments.

opts.GetValue( c, 0 ) value is one

$ parse ha:b:c:d:e: 0x01 -c one two
Prior to Options::Parse call:
  ParseOpts is ha:b:c:d:e:
  flag is 0x01
  argc is 3
  argv[ 0 ] is -c
  argv[ 1 ] is one
  argv[ 2 ] is two

opts.GetValue( c, 0 ) value is one
```

```
After Options::Parse call:
  argc is 1
  argv[ 0 ] is two
```

*Options::Parse(int &, StrPtr * &, const char *, int, const ErrorId &, Error *)*

Extract command line arguments and associated values.

Virtual?	No	
Class	`Options`	
Arguments	`int &argc`	Number of arguments
	`StrPtr * &argv`	An array of arguments to parse
	`const char *opts`	The list of valid options to extract
	`int flag`	A flag indicating how many arguments are expected to remain when parsing is complete
	`const ErrorId &usage`	An error message containing usage tips
	`Error *e`	The `Error` object to collect any errors encountered
Returns	`void`	

Notes

See the notes for the `char ** &argv` version of `Options::Parse()` for details.

See also

`Options::Parse()`

ServerHelperApi methods

This section lists the available ServerHelperApi methods.

ServerHelperApi::ClearProtocol()

Clear any protocol variables set using `ServerHelperApi::SetProtocol`

Virtual?	No
Class	`ServerHelperApi`
Arguments	None
Returns	`void`

Notes

`ClearProtocol()` unsets any protocol variables previously set using `SetProtocol()` or `SetProtocolV()`. Calling `ClearProtocol()`

ServerHelperApi::CloneFromRemote(int,const StrPtr*,ClientUser*,Error*)

Populates a personal server based on a remote specification.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`int depth`	specifies the number of revisions to fetch of each file
	`const StrPtr *df`	optional debug flag; set to 0 to leave it unset
	`ClientUser *ui`	a pointer to `ClientUser` object
	`Error *e`	a pointer to an `Error` object
Returns	`int`	indicates success of command

Notes

This method stores the remote generated by either `PrepareToCloneFilepath()` or `PrepareToCloneRemote()` as `origin` and then runs `p4 fetch` to populate the local server. This method should be run against the instance of `ServerHelperApi` class configured as the personal server.

A return value of 1 indicates a successful clone. If the method returns 0, check the `Error` object to see what occurred.

See also

`ServerHelperApi::PrepareToCloneFilepath()` `ServerHelperApi::PrepareToCloneRemote()`

Example

```
// Basic "p4 clone" flow:
ClientUser ui;
Error e;
ServerHelperApi personalServer( &e );

if( e.Test() )
    return 0;

personalServer.SetDvcsDir( "/path/to/root", &e );
personalServer.SetProg( "MyApp" ); // optional
personalServer.SetVersion( "version" ); // optional

if( personalServer.Exists() )
    return 0;

ServerHelperApi sharedServer( &e );
sharedServer.SetPort( "1666" );
sharedServer.SetProg( "MyApp" ); // optional
sharedServer.SetVersion( "version" ); // optional

// Fetch the remote spec
if( !personalServer.PrepareToCloneRemote( &sharedServer, remote, &ui )
)
   return 0;

// Create the personal server
// This returns the exit code from p4d, so 0 is success
if( personalServer.InitLocalServer( &ui ) )
    return 0;

// Fetch from the remote
```

```
if( !personalServer.CloneFromRemote( 0, 0, &ui ) )
    return 0;
```

ServerHelperApi::CloneFromRemote(int,const char*,ClientUser*,Error*)

Populates a personal server based on a remote specification.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`int depth`	specifies the number of revisions to fetch for each file
	`const char *df`	optional debug flag; set to 0 to leave it unset
	`ClientUser *ui`	a pointer to a `ClientUser` object
	`Error *e`	a pointer to an `Error` object
Returns	`int`	indicates success of command

Notes

See the notes for the `StrPtr*` version of `CloneFromRemote()` for details.

ServerHelperApi::CopyConfiguration (ServerHelperApi*,ClientUser*,Error*)

Copies server configuration information from a shared server in preparation for creating a matching personal server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`ServerHelperApi* remote`	the shared server to copy settings from
	`ClientUser *ui`	pointer to a `ClientUser` object
	`Error *e`	pointer to an `Error` object
Returns	`int`	

Notes

This method gets the case sensitivity and Unicode settings from a shared server. For a personal server to push and fetch against a shared server it must have matching case sensitivity and Unicode settings. Call this method prior to running `InitLocalServer()` to ensure that the new personal server will work properly with the shared server. This method can only be run if the personal server does not exist. Use the `Exists()` method to test to see if the personal server already exists.

Example

```
// Discover the case sensitivity and unicode settings
if( !personalServer.CopyConfiguration( &centralServer, &ui, &e ) )
    return 1;

// Create the local server
if( personalServer.InitLocalServer( &ui, &e ) )
    return 1;
```

ServerHelperApi::Exists(ClientUser,Error*)*

Checks for the existence of a personal server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`ClientUser * ui`	A `ClientUser` object
Returns	`int`	returns 1 if a personal server exists in the root dir

Notes

Many `ServerHelperApi` methods expect there to be no existing server in the path set by `SetDvcsDir()`. `Exists()` is useful for avoiding potential errors caused by pre-existing servers.

Example

```
// Check that we don't already have a DVCS tree at the target
directory
personalServer.SetDvcsDir( "/User/home/dev", &e );
if( e.Test() )
    return 1;
```

```
if( personalServer.Exists( &ui, &e ) )
    return 1;
```

ServerHelperApi::GetCaseFlag()

Gets the currently set case sensitivity flag value.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	None	
Returns	`StrPtr`	returns '-C0' or '-C1' or an empty StrPtr

Notes

Call after running `CopyConfiguration()` or `SetCaseFlag()` to see the value that will be used when initializing the personal server.

See also

`ServerHelperApi::GetUnicode()`

ServerHelperApi::GetClient()

Get the client workspace associated with this connection.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	None	
Returns	`const StrPtr&`	the current client workspace name

Notes

This method has the same behavior as the `ClientApi::GetClient()` method.

ServerHelperApi::GetClient(Error*)

Creates and initializes a ClientApi object for the specified server

Virtual?	No

Class	`ServerHelperApi`	
Arguments	`Error *e`	a pointer to an Error object
Returns	`ClientApi* client`	if successful, an initialized ClientApi object; otherwise 0

Notes

This is a convenience method to get a `ClientApi` object against the server associated with the `ServerHelperApi` object. `GetClient()` is the equivalent of creating a new `ClientApi` object, setting the `P4PORT`, `P4CLIENT`, and `P4USER`, and then running `ClientApi::Init()`. As with `ClientApi::Init()`, protocol variables must be set using `SetProtocol()` before running `GetClient()`.

See also

`ClientApi::Init()`

Example

```
// Create personal server
personalServer.CloneFromRemote( 0, 0, debug, &ui, &e )

// Turn on tagged output for the client
personalServer.SetProtocolV( P4Tag::v_tag );

// Get an init'ed ClientApi object
ClientApi *client = personalServer.GetClient( &e );
if( !client )
    return 1;

// Run 'p4 info'
client->RunTag( "info", &ui );
```

ServerHelperApi::GetDvcsDir()

Get the path to root of the personal server.

Virtual?	No

Class	`ServerHelperApi`	
Arguments	None	
Returns	`const StrPtr &`	the root path of the personal server

Notes

See `ClientApi::GetClient()` for more about the `StrPtr` return value.

ServerHelperApi::GetPort()

Get the connection information for this server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	None	
Returns	`const StrPtr &`	the hostname and port for the server

Notes

See `ClientApi::GetClient()` for more about the `StrPtr` return value.

ServerHelperApi::GetProg()

Get the application name set in the 'prog' variable.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	None	
Returns	`const StrPtr &`	the name of your application

Notes

See `ClientApi::GetClient()` for more about the `StrPtr` return value.

See also

`ServerHelperApi::SetProg()`

ServerHelperApi::GetQuiet()

Checks whether quiet output is enabled.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	None	
Returns	`int`	returns 1 if quiet output is enabled

Notes

If quiet is enabled, the output from clone and init are greatly reduced.

ServerHelperApi::GetUnicode()

Return the Unicode flag setting.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	None	
Returns	`int`	returns 1 if unicode is enabled

Notes

When a personal server is initialized it can either be in Unicode mode or not. If set to use Unicode mode all filenames and metadata will be translated from the code page used by the local machine into UTF-8. To be able to push or fetch, a personal server must match the Unicode setting of the shared server. Use the `CopyConfiguration()` method to copy the Unicode and case-sensitivity settings from the shared before initializing the personal server.

See also

`ServerHelperApi::GetCaseFlag()`

ServerHelperApi::GetUser()

Get the user name that is currently associated with the server connection.

Virtual?	No

Class	`ServerHelperApi`	
Arguments	None	
Returns	`const StrPtr &`	the active user name

Notes

See `ClientApi::GetClient()` for more about the `StrPtr` return value.

ServerHelperApi::GetVersion()

Get the application name set in the 'version' variable

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	None	
Returns	`const StrPtr &`	the version of your application

Notes

See `ClientApi::GetClient()` for more about the `StrPtr` return value.

See also

`ServerHelperApi::GetProg()`

ServerHelperApi::InitLocalServer(ClientUser,Error*)*

Creates the personal server based on configured settings.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`ClientUser *ui`	a ClientUser object
	`Error *e`	an Error object
Returns	`int`	returns the `p4d` return value; 0 indicates success

Notes

This method does the work of creating the personal server using the settings that have been previously set on the **`ServerHelperApi`** object. As part of creating the personal server it writes the **`P4CONFIG`** and **`P4IGNORE`** files, creates the **`.p4root`** dir, sets the case sensitivity and Unicode flags, creates the requisite serverId and server spec, and generates a protect table to restrict access to **`localhost`**.

If you are cloning from an existing shared server you will need to follow your call to **`InitLocalServer()`** with a call to **`CloneFromRemote()`**. **`InitLocalServer()`** only creates the personal server, it does not populate it.

Example

```
ClientUser ui;
Error e;
ServerHelperApi server( &e );

if( e.Test() )
    return 0;

server.SetDvcsDir( "/path/to/dvcs" );
server.SetProg( "MyApp" ); // optional
server.SetVersion( "version" ); // optional

if( server.Exists() )
    return 0;

// The unicode and case-sensitivity options must be set _before_
// InitLocalServer() is called. These can be set manually or
// discovered.

server.SetUnicode( true );
server.SetCaseSensitivity( "-C0" );

if( !server.InitLocalServer( &ui ) )
    return 0;
```

ServerHelperApi::PrepareToCloneFilepath (ServerHelperApi,const char*,ClientUser*,Error*)*

Creates a remote specification based on the provided filepath.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`ServerHelperApi *remote`	ServerHelperApi object representing the shared server
	`const char *path`	depot path to clone
	`Error *e`	an Error object
Returns	`int`	returns 1 if successful

Notes

This creates a new remote spec based on the provided filepath. It also checks that `p4 fetch` is allowed on the remote server specified.

See also

`ServerHelperApi::PrepareToCloneRemote()`

Example

```
ServerHelperApi remoteServer( &e );
remoteServer.SetPort( "1666" );
remoteServer.SetProg( "MyApp" ); // optional
remoteServer.SetVersion( "1.0" ); // optional

// Generate the remote spec
if( !localServer.PrepareToCloneFilepath( &remoteServer, path, &ui ) )
    return 0;

// Create the local server
// This returns the exit code from p4d, so 0 is success
if( localServer.InitLocalServer( &ui ) )
    return 0;
```

```
// Fetch from the remote
if( !localServer.CloneFromRemote( 0, 0, &ui ) )
    return 0;
```

ServerHelperApi::PrepareToCloneFilepath (ServerHelperApi,const StrPtr*,ClientUser*,Error*)*

Creates a remote specification based on the provided filepath.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`ServerHelperApi *remote`	ServerHelperApi object representing the shared server
	`const StrPtr *path`	depot path to clone
	`Error *e`	a pointer to an Error object
Returns	`int`	returns 1 if successful

Notes

See the `const char *` version of `PrepareToCloneFilepath()` for usage details.

ServerHelperApi::PrepareToCloneRemote (ServerHelperApi,const char*,ClientUser*,Error*)*

Creates a remote specification based on the provided remote from the shared server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`ServerHelperApi *remote`	ServerHelperApi object representing the shared server
	`const char *remote`	name of remote spec on shared server
	`Error *e`	a pointer to an Error object
Returns	`int`	returns 1 if successful

Notes

This creates a new remote spec based on the provided remote spec stored on the shared server. It also checks that `p4 fetch` is allowed on the remote server specified.

See also

`ServerHelperApi::PrepareToCloneFilepath()`

Example

```
ServerHelperApi remoteServer( &e );
remoteServer.SetPort( "1666" );
remoteServer.SetProg( "MyApp" ); // optional
remoteServer.SetVersion( "1.0" ); // optional

// Copy the remote spec from the shared server
if( !localServer.PrepareToCloneFilepath( &remoteServer, remote, &ui )
)
    return 0;

// Create the local server
// This returns the exit code from p4d, so 0 is success
if( localServer.InitLocalServer( &ui ) )
    return 0;

// Fetch from the remote
if( !localServer.CloneFromRemote( 0, 0, &ui ) )
    return 0;
```

ServerHelperApi::PrepareToCloneRemote (ServerHelperApi,const StrPtr*,ClientUser*,Error*)*

Creates a remote specification based on the provided remote from the shared server.

Virtual?	No
Class	`ServerHelperApi`

Arguments	`ServerHelperApi *remote`	ServerHelperApi object representing the shared server
	`const StrPtr *remote`	name of remote spec on shared server
	`Error *e`	a pointer to an Error object
Returns	`int`	returns 1 if successful

Notes

See the `const char *` version of `PrepareToCloneRemote()` for details.

See also

`ServerHelperApi::PrepareToCloneFilepath()`

ServerHelperApi::SetCaseFlag(const char,Error*)*

Sets the case flag for an uninitialized personal server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const char *flag`	the case sensitivity value
	`Error *e`	pointer to an Error object
Returns	`void`	

Notes

This sets the case sensitivity flag used when initializing a personal server. It can be set to '-C0' or '-C1'; '-C0' indicates the server is case-sensitive while '-C1' causes the server to be case-insensitive.

See also

`ServerHelperApi::SetUnicode()`

ServerHelperApi::SetCaseFlag(const StrPtr,Error*)*

Sets the case flag for an uninitialized personal server.

Virtual?	No

Class	`ServerHelperApi`	
Arguments	`const StrPtr *flag`	the case sensitivity value
	`Error *e`	pointer to an Error object
Returns	`void`	

Notes

See the `const char *` version of `GetCaseFlag()` for usage details.

ServerHelperApi::SetClient(const char)*

Sets the client workspace.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const char *name`	the name of the client workspace
Returns	`void`	

Notes

Sets the client workspace that should be used when connecting to the server.

ServerHelperApi::SetClient(const StrPtr)*

Sets the client workspace.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const StrPtr *name`	the name of the client workspace to use
Returns	`void`	

Notes

Sets the client workspace that should be used when connecting to the server.

ServerHelperApi::SetDefaultStream(const char)*

Sets the default stream name for the personal server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const char *name`	the name of the stream
Returns	`void`	

Notes

Sets the name of the stream to create when initializing the personal server. This is the equivalent of the '-c' flag when using the command line interface. Running `PrepareToCloneRemote()` will override the value set by `SetDefaultStream()`.

ServerHelperApi::SetDefaultStream(const StrPtr)*

Sets the default stream name for the personal server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const StrPtr *name`	the name of the stream
Returns	`void`	

Notes

See the `const char *` version of `SetDefaultStream()` for usage details.

ServerHelperApi::SetDvcsDir(const char)*

Sets the root dir for the personal server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const char *path`	full path to the root directory
	`Error *e`	an Error object
Returns	`int`	returns 0 if there is an error, 1 if successful

Notes

Sets the directory that will be used to hold the personal server. The directory should not already have a personal server in it or `InitLocalServer()` will fail. `SetDvcsDir()` does not check for an existing server; use the `Exists()` method after running `SetDvcsDir()` to test for an existing personal server.

ServerHelperApi::SetDvcsDir(const StrPtr)*

Sets the root dir for the personal server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const StrPtr *path`	full path to the root directory
	`Error *e`	an Error object
Returns	`int`	returns 0 if there is an error, 1 if successful

Notes

See the `const char *` version of `SetDvcsDir()` for usage details.

ServerHelperApi::SetPort(const char)*

Sets the server connection information.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const char *port`	the hostname and port to connect to
Returns	`void`	

Notes

Sets the hostname and port that will be connected to. It should be of the form 'hostname:port' such as 'helix.perforce.com:1666'.

ServerHelperApi::SetPort(const StrPtr)*

Sets the server connection information.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const StrPtr *name`	the hostname and port to connect to
Returns	`void`	

Notes

Sets the hostname and port that will be connected to. It should be of the form 'hostname:port' such as 'helix.perforce.com:1666'.

ServerHelperApi::SetProg(const char)*

Sets the program name for your application.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const char *name`	the name of your application
Returns	`void`	

Notes

Sets the program name for your application. This name will show up in the logs of any server your application runs against. Setting a program name makes it much easier to identify tools which are not running properly and is highly recommended.

ServerHelperApi::SetProg(const StrPtr)*

Sets the program name for your application.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const StrPtr *name`	the name of your application
Returns	`void`	

Notes

See the `const char *` version of `SetProg()` for usage details.

ServerHelperApi::SetProtocol(const char*,const char*)

Adds protocol tags for the `ServerHelperApi::GetClient()` method.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const char *p`	the protocol being set
	`const char *v`	the value to set
Returns	`void`	

Notes

See `SetProtocol()` on `ClientApi` for details.

ServerHelperApi::SetProtocolV(const char*)

Adds protocol tags for the `ServerHelperApi::GetClient()` method.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const char *p`	a single string with the protocol to set and the value
Returns	`void`	

Notes

See `SetProtocolV` on `ClientApi` for details.

ServerHelperApi::SetQuiet()

Enables reduced output for clone and init.

Virtual?	No
Class	`ServerHelperApi`
Arguments	none
Returns	`void`

Notes

Disables informational messages when running `CloneFromRemote()`.

ServerHelperApi::SetUnicode(int)

Sets or unsets the Unicode flag on the personal server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`int`	set to 1 to enable Unicode in the personal server
Returns	`void`	

Notes

This method is used to enable Unicode mode when initializing a personal server. This method needs to be called before calling `InitLocalServer()`. The Unicode setting of the personal server much match the Unicode setting of any shared servers it will push or fetch content to. `CopyConfiguration()` can be used to ensure the personal server matches the settings of the shared server.

ServerHelperApi::SetUser(const char)*

Sets the user account for connecting to the server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const char *name`	the user name
Returns	`void`	

Notes

Sets the user name for connecting to the associated server. The user must have permission to access any files being cloned from the shared server.

ServerHelperApi::SetUser(const StrPtr)*

Sets the user account for connecting to the server.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const StrPtr *name`	the user name to use
Returns	`void`	

Notes

See the `const char *` version of `SetUser()` for usage details.

ServerHelperApi::SetVersion(const char)*

Sets the version string for your application.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const char *version`	the version information for your application
Returns	`void`	

Notes

Sets the version string for your application. This version will show up in the logs of any server your application runs against. Setting a version makes it much easier to identify tools which are not running properly and is highly recommended.

See also

`ServerHelperApi::SetProg()`

ServerHelperApi::SetVersion(const StrPtr)*

Sets the version string for your application.

Virtual?	No	
Class	`ServerHelperApi`	
Arguments	`const StrPtr *name`	the version information for your application
Returns	`void`	

Notes

See the `const char *` version of `SetVersion()` for usage details.

Signaler methods

This section lists the available Signaler methods.

Signaler::Block()

Cause interrupt signals from the user to be ignored until a subsequent call to `Signaler::Catch()`.

Virtual?	No
Class	`Signaler`
Arguments	None
Returns	`void`

Notes

`Block()` does not actually block the signals, but causes the process to ignore them.

For portability reasons, `Block()` and `Catch()` use the BSD/ANSI C `signal(2)` function rather than the POSIX `sigaction()`.

See also

`Signaler::Catch()` `Signaler::OnIntr()`

Example

```
#include <unistd.h>  // for sleep()
#include <stdhdrs.h>
#include <strbuf.h>
#include <signaler.h>

int main( int argc, char **argv )
{
    // Block ^C
    printf( "For the next 5 seconds, ^C will be ignored\n" );
    signaler.Block();
    sleep( 5 );
```

```
    printf( "Enabling ^C again\n" );
    signaler.Catch();
    for ( ; ; )
        sleep( 60 );
    exit( 0 );
}
```

Signaler::Catch()

Allow interrupt signals from the user to be delivered once more following a previous call to `Signaler::Block()`.

Virtual?	No
Class	`Signaler`
Arguments	None
Returns	`void`

Notes

`Catch()` does not replace your signal handler if you have already replaced the `Signaler` class' handler with one of your own using the ANSI `signal(2)` function.

For portability reasons, `Block()` and `Catch()` use the BSD/ANSI C `signal(2)` function rather than the POSIX `sigaction()`.

See also

`Signaler::Block()` `Signaler::OnIntr()`

Example

```
int main( int argc, char **argv )
{
    // Block ^C
    printf( "For the next 5 seconds, ^C will be ignored\n" );
    signaler.Block();
    sleep( 5 );

    printf( "Enabling ^C again\n" );
```

```
    signaler.Catch();
    for ( ; ; )
        sleep( 60 );
    exit( 0 );
}
```

Signaler::DeleteOnIntr(void *)

Removes a function previously registered using `OnIntr()` from the list.

Virtual?	No	
Class	`Signaler`	
Arguments	`void *ptr`	Pointer to the data item with which the original function was registered
Returns	`void`	

See also

`Signaler::OnIntr()` `Signaler::Intr()`

Example

```
#include <unistd.h>    // for sleep()
#include <stdhdrs.h>
#include <strbuf.h>
#include <signaler.h>

class MyClass
{
    public:
    void         Set( StrPtr *d ) { data = *d; }
    const StrPtr *Get()           { return &data; }
    void         Identify()       { printf( "I'm %s\n", data.Text() );
}

    private:
    StrBuf       data;
```

```
};

static void InterruptHandler( void *p )
{
    MyClass      *m = ( MyClass * )p;
    m->Identify();
}

int main( int argc, char **argv )
{
    StrBuf   data;
    MyClass  *list[ 5 ];

    for ( int i = 1; i <= 5; i++ )
    {
        data.Set( "Object" );
        data << i;

        MyClass   *p = new MyClass;
        list[ i - 1 ] = p;

        p->Set( &data );

        signaler.OnIntr( InterruptHandler, (void *)p );
    }

    // Unregister Object 3
    signaler.DeleteOnIntr( list[ 2 ] );

    printf( "Hit ^C to fire the interrupt handler\n" );
    for ( ; ; )
        sleep( 60 );

    exit( 0 );
}
```

Signaler::Intr()

Coordinate execution of all functions registered by `Signaler::OnIntr()`.

Virtual?	No
Class	`Signaler`
Arguments	None
Returns	`void`

Notes

`Intr()` is the `Signaler` class's main handler for interrupt signals.

Most Helix server client applications do not need to call `Intr()` directly, because it is called directly from the internal handler function that catches the interrupt signals.

This internal handler function also causes the process to exit, returning an exit status of -1 to the operating system. (For instance, `signaler.Intr(); exit( -1 )`)

If you require more flexible or complex interrupt handling, replace the default interrupt handler function with your own by using the ANSI C `signal(2)` function, and call `Intr()` to execute the registered functions.

Caveat

`Intr()` does not deregister functions after they have been called. When calling a registered function twice might cause a failure, immediately deregister it using `DeleteOnIntr()` after the function has been called.

See also

`Signaler::OnIntr()`

Example

```
#include <unistd.h>    // for sleep()
#include <signal.h>
#include <stdhdrs.h>
#include <strbuf.h>
#include <signaler.h>

class MyClass
{
```

```
    public:
    void         Set( StrPtr *d ) { data = *d; }
    const StrPtr *Get()           { return &data; }
    void         Identify()       { printf( "I'm %s\n", data.Text() );
}

    private:
    StrBuf       data;
};

static int intrCount = 0;
static const int maxIntr = 3;

// Replacement handler for SIGINT signals. Overrides Signaler class's
// default handler to avoid immediate exit.

static void trap_interrupt( int sig )
{
    intrCount++;
    printf( "Received SIGINT. Calling registered functions...\n" );
    signaler.Intr();
    printf( "All functions done\n\n" );
    if ( intrCount >= maxIntr )
    {
        printf( "Interrupt limit hit. Exiting...\n" );
        exit( 0 );
    }
}

static void InterruptHandler( void *p )
{
    MyClass      *m = ( MyClass * )p;
    m->Identify();

    // Don't identify this object again
```

```
    signaler.DeleteOnIntr( p );
}

int main( int argc, char **argv )
{
    signal( SIGINT, trap_interrupt );
    signaler.Catch();
```

```
    int objCount = 5;
    int nextId = 1;
    for ( ; ; )
    {
        int i;
        for ( i = nextId; i < nextId + objCount; i++ )
        {
            StrBuf data;

            data.Set( "Object" );
            data << i;

            MyClass *p = new MyClass;
            p->Set( &data );

            printf( "Registering %s\n", data.Text() );
            signaler.OnIntr( InterruptHandler, ( void * )p );
        }

        nextId = i;
        printf( "\n" );
        printf( "Hit ^C to fire the interrupt handler [%d to go]\n",
                maxIntr - intrCount );
        sleep( 10 );
    }

    exit( 0 );
}
```

*Signaler::OnIntr(SignalFunc, void *)*

Register a function and argument to be called when an interrupt signal is received.

Virtual?	No	
Class	`Signaler`	
Arguments	`SignalFunc callback`	Pointer to a function to call on receipt of an interrupt signal. The function must have the prototype `voidfunc( void *ptr )`
	`void *ptr`	Pointer to a data item to pass to the callback function when invoking it.
Returns	`void`	

Notes

Functions are called in the reverse order that they are registered.

See also

`Signaler::DeleteOnIntr()` `Signaler::Intr()`

Example

```
#include <unistd.h>     // for sleep()
#include <stdhdrs.h>
#include <strbuf.h>
#include <signaler.h>

class MyClass
{
    public:
    void         Set( StrPtr *d ) { data = *d; }
    const StrPtr *Get()           { return &data; }
    void         Identify()       { printf( "I'm %s\n", data.Text() );
}

    private:
    StrBuf       data;
```

```
};

static void InterruptHandler( void *p )
{
    MyClass      *m = ( MyClass * )p;
    m->Identify();
}

int main( int argc, char **argv )
{
    for ( int i = 1; i <= 5; i++ )
    {
        StrBuf data;

        data.Set( "Object" );
        data << i;

        MyClass *p = new MyClass;
        p->Set( &data );

        signaler.OnIntr( InterruptHandler, ( void * )p );
    }

    printf( "Hit ^C to fire the interrupt handler\n" );
    for ( ; ; )
        sleep( 60 );

    exit( 0 );
}
```

Signaler::Signaler() (constructor)

Constructs a new **`Signaler`** object.

Virtual?	No

Class	`Signaler`
Arguments	N/A
Returns	N/A

Notes

There is rarely a need for API users to construct `Signaler` objects themselves. Use the global `Signaler` variable `signaler` instead.

See also

`Signaler::OnIntr()` `Signaler::DeleteOnIntr()`

StrBuf methods

This section lists the available StrBuf methods.

StrBuf::Alloc(int)

Allocate an additional specified number of bytes to a `StrBuf`. The string pointed to by the `StrBuf`'s `buffer` is logically extended.

Virtual?	No	
Class	`StrBuf`	
Arguments	`int len`	number of bytes to be allocated
Returns	`char *`	pointer to the first additional byte allocated

Notes

The `length` of the `StrBuf` is incremented by the `len` argument.

If the memory for the `StrBuf`'s `buffer` is not large enough, enough new memory is allocated to contiguously contain the extended string. If new memory is allocated, the old memory is freed. (All `StrBuf` member functions with the potential to increase the length of a `StrBuf` manage memory this way.)

A call to `Alloc()` might change the string pointed to by the `StrBuf`'s `buffer`; do not rely on pointer arithmetic to determine the new pointer, because the call to `Alloc()` might have moved the buffer location.

Example

```
#include <iostream>
#include <iomanip>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf sb;
    char *p;

    sb.Set( "xyz" );

    cout << "sb.Text() prior to sb.Alloc( 70 ) returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "(int)sb.Text() prior to sb.Alloc( 70 ) returns 0x" <<
hex;
    cout << setw( 8 ) << setfill( '0' ) << (int)sb.Text() << dec <<
"\n";
    cout << "sb.Length() prior to sb.Alloc( 70 ) returns ";
    cout << sb.Length() << "\n\n";

    p = sb.Alloc( 70 );   // allocate in StrBuf

    cout << "sb.Text() after sb.Alloc( 70 ) returns (first three
bytes) ";
    cout << "\"" << setw( 3 ) << sb.Text() << "\"\n";
    cout << "(int)sb.Text() after sb.Alloc( 70 ) returns 0x" << hex;
    cout << setw( 8 ) << setfill( '0' ) << (int)sb.Text() << dec <<
"\n";
    cout << "(int)sb.Alloc( 70 ) returned 0x" << hex;
    cout << setw( 8 ) << setfill( '0' ) << (int)p << dec << "\n";
    cout << "sb.Length() after sb.Alloc( 70 ) returns ";
```

```
    cout << sb.Length() << "\n";
}
```

Executing the preceding code produces the following output:

```
sb.Text() prior to sb.Alloc( 70 ) returns "xyz"
(int)sb.Text() prior to sb.Alloc( 70 ) returns 0x0804a9a0
sb.Length() prior to sb.Alloc( 70 ) returns 3

sb.Text() after sb.Alloc( 70 ) returns (first three bytes) "xyz"
(int)sb.Text() after sb.Alloc( 70 ) returns 0x0804a9b0
(int)sb.Alloc( 70 ) returned 0x0804a9b3
sb.Length() after sb.Alloc( 70 ) returns 73
```

*StrBuf::Append(const char *)*

Append a null-terminated string to a **`StrBuf`**. The string is logically appended to the string pointed to by the **`StrBuf`**'s **`buffer`**.

Virtual?	No	
Class	**`StrBuf`**	
Arguments	**`const char *buf`**	pointer to the first byte of the null-terminated string
Returns	**`void`**	

Notes

The **`StrBuf`**'s **`length`** is incremented by the number of bytes prior to the first null byte in the string.

If the memory for the **`StrBuf`**'s **`buffer`** is not large enough, new memory to contiguously contain the results of appending the null-terminated string is allocated. If new memory is allocated, the old memory is freed. Any memory allocated is separate from the memory for the string.

Example

```
int main( int argc, char **argv )
{
    char chars[] = "zy";
    StrBuf sb;

    sb.Set( "xyz" );
```

```
    cout << "sb.Text() prior to sb.Append( chars ) returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() prior to sb.Append( chars ) returns ";
    cout << sb.Length() << "\n\n";

    sb.Append( chars );   // append char * to StrBuf

    cout << "sb.Text() after sb.Append( chars ) returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() after sb.Append( chars ) returns ";
    cout << sb.Length() << "\n";
}
```

Executing the preceding code produces the following output:

```
sb.Text() prior to sb.Append( chars ) returns "xyz"
sb.Length() prior to sb.Append( chars ) returns 3

sb.Text() after sb.Append( chars ) returns "xyzzy"
sb.Length() after sb.Append( chars ) returns 5
```

*StrBuf::Append(const char *, int)*

Append a string of a specified length to a `StrBuf`. The string is logically appended to the string pointed to by the `StrBuf`'s `buffer`.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const char *buf`	pointer to the first byte of the string
	`int len`	length of the string
Returns	`void`	

Notes

Exactly `len` bytes are appended to the `StrBuf` from the string. The `length` of the `StrBuf` is incremented by the `len` argument.

If the memory for the **StrBuf**'s **buffer** is not large enough, new memory to contiguously contain the results of appending the string of specified length is allocated. If new memory is allocated, the old memory is freed. Any memory allocated is separate from the memory for the string.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    char chars[] = "zyx";
    StrBuf sb;

    sb.Set( "xyz" );

    cout << "sb.Text() prior to sb.Append( chars, 2 ) returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() prior to sb.Append( chars, 2 ) returns ";
    cout << sb.Length() << "\n\n";

    sb.Append( chars, 2 );   // append len bytes of char * to StrBuf

    cout << "sb.Text() after sb.Append( chars, 2 ) returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() after sb.Append( chars, 2 ) returns ";
    cout << sb.Length() << "\n";
}
```

Executing the preceding code produces the following output:

```
sb.Text() prior to sb.Append( chars, 2 ) returns "xyz"
sb.Length() prior to sb.Append( chars, 2 ) returns 3

sb.Text() after sb.Append( chars, 2 ) returns "xyzzy"
sb.Length() after sb.Append( chars, 2 ) returns 5
```

StrBuf::Append(const StrPtr *)

Append a `StrPtr` to a `StrBuf`. The argument is passed as a pointer to the `StrPtr`. The string pointed to by the `StrPtr`'s `buffer` is logically appended to the string pointed to by the `StrBuf`'s `buffer`. Arguments are commonly addresses of instances of classes derived from the `StrPtr` class, such as `StrRef` and `StrBuf`.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const StrPtr *s`	pointer to the `StrPtr` instance
Returns	`void`	

Notes

Initialize the `StrBuf` and the `StrPtr` before calling `Append()`.

Exactly the number of bytes specified by the `length` of the `StrPtr` are appended to the `StrBuf` from the `StrPtr`. The `length` of the `StrBuf` is incremented by the `length` of the `StrPtr`.

If the memory for the `StrBuf`'s `buffer` is not large enough, new memory to contiguously contain the results of appending the `StrPtr` is allocated. If new memory is allocated, the old memory is freed. Any memory allocated is separate from the memory for the `StrPtr`.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrRef sr( "zy" );
    StrPtr *sp = &sr;
    StrBuf sba;
    StrBuf sbb;

    sba.Set( "xyz" );
    sbb.Set( "xyz" );
```

```
    cout << "sba.Text() after sba.Set( \"xyz\" ) returns ";
    cout << "\"" << sba.Text() << "\"\n";
    cout << "sba.Length() after sba.Set( \"xyz\" ) returns ";
    cout << sba.Length() << "\n";
    cout << "sbb.Text() after sbb.Set( \"xyz\" ) returns ";
    cout << "\"" << sbb.Text() << "\"\n";
    cout << "sbb.Length() after sbb.Set( \"xyz\" ) returns ";
    cout << sbb.Length() << "\n\n";

    sba.Append( sp );   // append StrPtr * to StrBuf

    cout << "sba.Text() after sba.Append( sp ) returns ";
    cout << "\"" << sba.Text() << "\"\n";
    cout << "sba.Length() after sba.Append( sp ) returns ";
    cout << sba.Length() << "\n\n";

    sbb.Append( &sr );   // append &StrRef to StrBuf

    cout << "sbb.Text() after sbb.Append( &sr ) returns ";
    cout << "\"" << sbb.Text() << "\"\n";
    cout << "sbb.Length() after sbb.Append( &sr ) returns ";
    cout << sbb.Length() << "\n\n";

    sba.Append( &sbb );   // append &StrBuf to StrBuf

    cout << "sba.Text() after sba.Append( &sbb ) returns ";
    cout << "\"" << sba.Text() << "\"\n";
    cout << "sba.Length() after sba.Append( &sbb ) returns ";
    cout << sba.Length() << "\n";
}
```

Executing the preceding code produces the following output:

```
sba.Text() after sba.Set( "xyz" ) returns "xyz"
sba.Length() after sba.Set( "xyz" ) returns 3
sbb.Text() after sbb.Set( "xyz" ) returns "xyz"
sbb.Length() after sbb.Set( "xyz" ) returns 3
```

```
sba.Text() after sba.Append( sp ) returns "xyzzy"
sba.Length() after sba.Append( sp ) returns 5
sbb.Text() after sbb.Append( &sr ) returns "xyzzy"
sbb.Length() after sbb.Append( &sr ) returns 5
sba.Text() after sba.Append( &sbb ) returns "xyzzyxyzzy"
sba.Length() after sba.Append( &sbb ) returns 10
```

StrBuf::Clear()

Clear the `length` member of a `StrBuf`.

Virtual?	No
Class	`StrBuf`
Arguments	None
Returns	`void`

Notes

Only the `length` member of the `StrBuf` is zeroed.

To set the `buffer` member to a zero-length string, call `Terminate()` after calling `Clear()`.

See also

`StrBuf::Terminate()`

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf sb;

    sb.Set( "xyz" );
```

```
    cout << "Prior to sb.Clear() and sb.Terminate():\n";
    cout << "  sb.Length() returns " << sb.Length() << "\n";
    cout << "  sb.Text() returns \"" << sb.Text() << "\"\n\n";

    sb.Clear();   // zero out the length

    cout << "After sb.Clear() but prior to sb.Terminate():\n";
    cout << "  sb.Length() returns " << sb.Length() << "\n";
    cout << "  sb.Text() returns \"" << sb.Text() << "\"\n\n";

    sb.Terminate();

    cout << "After sb.Clear() and sb.Terminate():\n";
    cout << "  sb.Length() returns " << sb.Length() << "\n";
    cout << "  sb.Text() returns \"" << sb.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
Prior to sb.Clear() and sb.Terminate():
  sb.Length() returns 3
  sb.Text() returns "xyz"

After sb.Clear() but prior to sb.Terminate():
  sb.Length() returns 0
  sb.Text() returns "xyz"

After sb.Clear() and sb.Terminate():
  sb.Length() returns 0
  sb.Text() returns ""
```

StrBuf::StrBuf() (Constructor)

Construct a `StrBuf`.

Virtual?	No

Class	`StrBuf`
Arguments	None

Notes

The `StrBuf` constructor initializes the `StrBuf` to contain a zero-length null buffer.

Example

```
int main( int argc, char **argv )
{
    StrBuf sb;   // constructor called

    cout << "sb.Text() returns \"" << sb.Text() << "\"\n";
    cout << "sb.Length() returns " << sb.Length() << "\n";
}
```

Executing the preceding code produces the following output:

```
sb.Text() returns ""
sb.Length() returns 0
```

StrBuf::StrBuf(const StrBuf &) (Copy Constructor)

Construct a copy of a `StrBuf`.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const StrBuf &s`	(implied) reference of the `StrBuf` from which copying occurs

Notes

The `StrBuf` copy constructor creates a copy of a `StrBuf`. The `StrBuf` from which copying occurs must be initialized before calling the copy constructor.

The `StrBuf` copy constructor initializes the new `StrBuf` to contain a zero-length null buffer, and sets the contents of the new `StrBuf` using the contents of the original `StrBuf`. Any memory allocated for the `buffer` of the copy is separate from the memory for the `buffer` of the original `StrBuf`.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

void called( StrBuf csb )
{
    csb << "zy";

    cout << "called() csb.Text() returns \"" << csb.Text() << "\"\n";
}

int main( int argc, char **argv )
{
    StrBuf sb;
    sb.Set( "xyz" );
    called( sb );   // copy constructor called
    cout << "main() sb.Text() returns \"" << sb.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
called() csb.Text() returns "xyzzy"
main() sb.Text() returns "xyz"
```

StrBuf::~StrBuf() (Destructor)

Destroy a **`StrBuf`**.

Virtual?	No
Class	**`StrBuf`**
Arguments	None
Returns	N/A

Notes

The **`StrBuf`** destructor destroys a **`StrBuf`**.

If the **buffer** points to allocated memory other than **nullStrBuf**, the allocated memory is freed.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf *psb;
    psb = new StrBuf;
    psb->Set( "xyz" );
    cout << psb- → Text() returns \" << psb->Text() << "\"\n";
    delete psb;   // destructor called and allocated memory freed
}
```

Executing the preceding code produces the following output:

```
psb->Text() returns "xyz"
```

StrBuf::Extend(char)

Extend a **StrBuf** by one byte. The string pointed to by the **StrBuf**'s **buffer** is logically extended.

Virtual?	No	
Class	**StrBuf**	
Arguments	**char c**	the byte copied to the extended string
Returns	**void**	

Notes

One byte is copied to the extended **StrBuf**. The **length** of the **StrBuf** is incremented by one.

Extend() does not null-terminate the extended string pointed to by the **StrBuf**'s **buffer**. To ensure that the extended string is null-terminated, call **Terminate()** after calling **Extend()**.

If the memory for the **StrBuf**'s **buffer** is not large enough, enough new memory is allocated to contiguously contain the extended string. If new memory is allocated, the old memory is freed. Any memory allocated is separate from the memory for the byte.

See also

StrBuf::Terminate()

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf sb;

    sb.Set( "xy" );

    cout << "sb.Text() prior to sb.Extend( 'z' ) returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() prior to sb.Extend( 'z' ) returns ";
    cout << sb.Length() << "\n\n";

    sb.Extend( 'z' );   // extend StrBuf from char
    sb.Terminate();

    cout << "sb.Text() after sb.Extend( 'z' ) returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() after sb.Extend( 'z' ) returns ";
    cout << sb.Length() << "\n";
}
```

Executing the preceding code produces the following output:

```
sb.Text() prior to sb.Extend( 'z' ) returns "xy"
sb.Length() prior to sb.Extend( 'z' ) returns 2

sb.Text() after sb.Extend( 'z' ) returns "xyz"
sb.Length() after sb.Extend( 'z' ) returns 3
```

StrBuf::Extend(const char *, int)

Extend a `StrBuf` by a string of a specified length. The string pointed to by the `StrBuf`'s `buffer` is logically extended.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const char *buf`	pointer to the first byte of the string
	`int len`	length of the string
Returns	`void`	

Notes

Exactly `len` bytes are copied from the string to the extended `StrBuf`. The `length` of the `StrBuf` is incremented by `len` bytes.

`Extend()` does not null-terminate the extended string pointed to by the `StrBuf`'s `buffer`. To ensure that the extended string is null-terminated, call `Terminate()` after calling `Extend()`.

If the memory for the `StrBuf`'s `buffer` is not large enough, enough new memory is allocated to contiguously contain the extended string. If new memory is allocated, the old memory is freed. Any memory allocated is separate from the memory for the string.

See also

`StrBuf::Terminate()`

Example

```
int main( int argc, char **argv )
{
    char chars[] = "zyx";
    StrBuf sb;

    sb.Set( "xyz" );

    cout << "sb.Text() prior to sb.Extend( chars, 2 ) returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() prior to sb.Extend( chars, 2 ) returns ";
    cout << sb.Length() << "\n\n";
```

```
    sb.Extend( chars, 2 );   // extend StrBuf from len bytes of char *
    sb.Terminate();

    cout << "sb.Text() after sb.Extend( chars, 2 ) returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() after sb.Extend( chars, 2 ) returns ";
    cout << sb.Length() << "\n";
}
```

Executing the preceding code produces the following output:

```
sb.Text() prior to sb.Extend( chars, 2 ) returns "xyz"
sb.Length() prior to sb.Extend( chars, 2 ) returns 3
sb.Text() after sb.Extend( chars, 2 ) returns "xyzzy"
sb.Length() after sb.Extend( chars, 2 ) returns 5
```

*StrBuf::operator =(const char *)*

Assign a `StrBuf` from a null-terminated string.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const char *buf`	(implied) pointer to the first byte of the null-terminated string
Returns	`void`	

Notes

Initialize the `StrBuf` before the assignment.

The `length` is set to the number of bytes prior to the first null byte in the string.

Any memory allocated for the `StrBuf`'s `buffer` is separate from the memory for the string.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>
```

```
int main( int argc, char **argv )
{
    char chars[] = "xyz";
    StrBuf sb;

    sb = chars;   // assign StrBuf from char *

    cout << "chars[] = \"" << chars << "\"\n";
    cout << "sb.Text() returns \"" << sb.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
chars[] = "xyz"
sb.Text() returns "xyz"
```

StrBuf::operator =(const StrBuf &)

Assign a `StrBuf` from another `StrBuf`.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const StrBuf &buf`	(implied) reference of the `StrBuf` from which assignment occurs
Returns	`void`	

Notes

Initialize both `StrBuf`s before the assignment.

Any memory allocated for the assigned `StrBuf`'s `buffer` is separate from the memory for the `StrBuf`'s `buffer` from which assignment occurs.

Do not assign a `StrBuf` to itself.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>
```

```
int main( int argc, char **argv )
{
    StrBuf sba;
    StrBuf sbb;

    sba.Set( "xyz" );

    sbb = sba;   // assign StrBuf to StrBuf

    cout << "sba.Text() returns \"" << sba.Text() << "\"\n";
    cout << "sbb.Text() returns \"" << sbb.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
sba.Text() returns "xyz"
sbb.Text() returns "xyz"
```

StrBuf::operator =(const StrPtr &)

Assign a `StrBuf` from a `StrPtr`.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const StrPtr &s`	(implied) reference of the `StrPtr` instance
Returns	`void`	

Notes

Initialize the `StrBuf` and the `StrPtr` before assignment.

Any memory allocated for the `StrBuf`'s `buffer` is separate from the memory for the `StrPtr`'s `buffer`.

Example

```
#include <iostream>

#include <stdhdrs.h>
```

```
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrRef sr( "xyz" );
    StrPtr *sp = &sr;
    StrBuf sb;

    sb = *sp;   // assign StrBuf from StrPtr

    cout << sp- → Text() returns \" << sp->Text() << "\"\n";
    cout << "sb.Text() returns \"" << sb.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
sp->Text() returns "xyz"
sb.Text() returns "xyz"
```

StrBuf::operator =(const StrRef &)

Assign a **`StrBuf`** from a **`StrRef`**.

Virtual?	No	
Class	**`StrBuf`**	
Arguments	**`const StrRef &s`**	(implied) reference of the **`StrRef`** instance
Returns	**`void`**	

Notes

Initialize the **`StrBuf`** and **`StrRef`** before assignment.

Any memory allocated for the **`StrBuf`**'s **`buffer`** is separate from that of the **`StrRef`**'s **`buffer`**.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>
```

```
int main( int argc, char **argv )
{
    StrRef sr( "xyz" );
    StrBuf sb;

    sb = sr;   // assign StrBuf from StrRef

    cout << "sr.Text() returns \"" << sr.Text() << "\"\n";
    cout << "sb.Text() returns \"" << sb.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
sr.Text() returns "xyz"
sb.Text() returns "xyz"
```

*StrBuf::operator <<(const char *)*

Append a null-terminated string to a **`StrBuf`**. The string is logically appended to the string pointed to by the **`StrBuf`**'s **`buffer`**.

Virtual?	No	
Class	**`StrBuf`**	
Arguments	**`const char *s`**	(implied) pointer to the first byte of the null-terminated string
Returns	**`StrBuf &`**	reference of the **`StrBuf`**

Notes

The **`StrBuf`**'s **`length`** is incremented by the number of bytes prior to the first null byte in the string.

If the memory for the **`StrBuf`**'s **`buffer`** is not large enough, new contiguous memory is allocated to contain the results of appending the null-terminated string. If new memory is allocated, the old memory is freed. Any memory allocated is separate from the memory for the string.

Example

```
#include <iostream>

#include <stdhdrs.h>
```

```
#include <strbuf.h>

int main( int argc, char **argv )
{
    char chars[] = "zy";
    StrBuf sb;

    sb.Set( "xyz" );

    cout << "sb.Text() prior to sb << chars returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() prior to sb << chars returns ";
    cout << sb.Length() << "\n\n";

    sb << chars;   // append char * to StrBuf

    cout << "sb.Text() after sb << chars returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() after sb << chars returns ";
    cout << sb.Length() << "\n";
}
```

Executing the preceding code produces the following output:

```
sb.Text() prior to sb << chars returns "xyz"
sb.Length() prior to sb << chars returns 3

sb.Text() after sb << chars returns "xyzzy"
sb.Length() after sb << chars returns 5
```

StrBuf::operator <<(int)

Append a formatted integer to a **`StrBuf`**. The formatted integer is logically appended to the string pointed to by the **`StrBuf`**'s **`buffer`**.

Virtual?	No
Class	**`StrBuf`**

Arguments	`int v`	(implied) integer
Returns	`StrBuf &`	reference of the `StrBuf`

Notes

The integer is formatted with the logical equivalent of `sprintf( buf, "%d", v )`.

The `length` is incremented by the number of bytes of the formatted integer.

If the memory for the `StrBuf`'s `buffer` is not large enough, new contiguous memory is allocated to contain the results of appending the formatted integer. If new memory is allocated, the old memory is freed. Any memory allocated is separate from the memory for the formatted integer.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf sb;
    int i;

    sb.Set( "xyz" );
    i = 73;

    cout << "sb.Text() prior to sb << i returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() prior to sb << i returns ";
    cout << sb.Length() << "\n\n";

    sb << i;   // append (formatted) int to StrBuf

    cout << "sb.Text() after sb << i returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() after sb << i returns ";
```

```
    cout << sb.Length() << "\n";
}
```

Executing the preceding code produces the following output:

```
sb.Text() prior to sb << i returns "xyz"
sb.Length() prior to sb << i returns 3

sb.Text() after sb << i returns "xyz73"
sb.Length() after sb << i returns 5
```

StrBuf::operator <<(const StrPtr *)

Append a `StrPtr` to a `StrBuf`. The string pointed to by the `StrPtr`'s `buffer` is logically appended to the string pointed to by the `StrBuf`'s `buffer`.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const StrPtr *s`	(implied) pointer to the `StrPtr` instance
Returns	`StrBuf &`	reference of the `StrBuf`

Notes

Exactly the number of bytes specified by the `StrPtr`'s `length` are appended to the `StrBuf`. The `StrBuf`'s `length` is incremented by the `StrPtr`'s `length`.

If the memory for the `StrBuf`'s `buffer` is not large enough, new contiguous memory is allocated to contain the results of appending the `StrPtr`. If new memory is allocated, the old memory is freed. Any memory allocated is separate from the memory for the `StrPtr`.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrRef sr( "zy" );
    StrPtr *sp = &sr;
```

```
    StrBuf sb;

    sb.Set( "xyz" );

    cout << "sb.Text() prior to sb << sp returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() prior to sb << sp returns ";
    cout << sb.Length() << "\n\n";

    sb << sp;   // append StrPtr * to StrBuf

    cout << "sb.Text() after sb << sp returns ";
    cout << "\"" << sb.Text() << "\"\n";
    cout << "sb.Length() after sb << sp returns ";
    cout << sb.Length() << "\n";
}
```

Executing the preceding code produces the following output:

```
sb.Text() prior to sb << sp returns "xyz"
sb.Length() prior to sb << sp returns 3

sb.Text() after sb << sp returns "xyzzy"
sb.Length() after sb << sp returns 5
```

StrBuf::operator <<(const StrPtr &)

Append a `StrPtr` to a `StrBuf`. The argument is passed as a reference of the `StrPtr`. The string pointed to by the `StrPtr`'s `buffer` is logically appended to the string pointed to by the `StrBuf`'s `buffer`.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const StrPtr &s`	(implied) reference of the `StrPtr` instance
Returns	`StrBuf &`	reference of the `StrBuf`

Notes

Arguments are typically instances of classes derived from the **StrPtr** class, such as **StrRef** and **StrBuf**.

Exactly the number of bytes specified by the **length** of the **StrPtr** are appended to the **StrBuf** from the **StrPtr**. The **length** of the **StrBuf** is incremented by the **length** of the **StrPtr**.

If the memory for the **StrBuf**'s **buffer** is not large enough, new contiguous memory is allocated to contain the results of appending the **StrPtr**. If new memory is allocated, the old memory is freed. Any memory allocated is separate from the memory for the **StrPtr**.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrRef sr( "zy" );
    StrPtr *sp = &sr;
    StrBuf sba;
    StrBuf sbb;

    sba.Set( "xyzzy" );
    sbb.Set( "xyz" );

    cout << "sba.Text() after sba.Set( \"xyzzy\" ) returns ";
    cout << "\"" << sba.Text() << "\"\n";
    cout << "sba.Length() after sba.Set( \"xyzzy\" ) returns ";
    cout << sba.Length() << "\n";
    cout << "sbb.Text() after sbb.Set( \"xyz\" ) returns ";
    cout << "\"" << sbb.Text() << "\"\n";
    cout << "sbb.Length() after sbb.Set( \"xyz\" ) returns ";
    cout << sbb.Length() << "\n";

    sbb << sr;   // append StrRef to StrBuf
```

```
    cout << "sbb.Text() after sbb << sr returns ";
    cout << "\"" << sbb.Text() << "\"\n";
    cout << "sbb.Length() after sbb << sr returns ";
    cout << sbb.Length() << "\n";

    sba << sbb;   // append StrBuf to StrBuf

    cout << "sba.Text() after sba << sbb returns ";
    cout << "\"" << sba.Text() << "\"\n";
    cout << "sba.Length() after sba << sbb returns ";
    cout << sba.Length() << "\n";
}
```

Executing the preceding code produces the following output:

```
sba.Text() after sba.Set( "xyzzy" ) returns "xyzzy"
sba.Length() after sba.Set( "xyzzy" ) returns 5
sbb.Text() after sbb.Set( "xyz" ) returns "xyz"
sbb.Length() after sbb.Set( "xyz" ) returns 3
sbb.Text() after sbb << sr returns "xyzzy"
sbb.Length() after sbb << sr returns 5
sba.Text() after sba << sbb returns "xyzzyxyzzy"
sba.Length() after sba << sbb returns 10
```

*StrBuf::Set(const char *)*

Set a `StrBuf` from a null-terminated string.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const char *buf`	pointer to the first byte of the null-terminated string
Returns	`void`	

Notes

Initialize the `StrBuf` before calling `Set()`.

The `length` of the `StrBuf` is set to the number of bytes prior to the first null byte in the string.

Any memory allocated for the `StrBuf`'s `buffer` is separate from the memory for the string.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    char chars[] = "string";
    StrBuf sb;

    sb.Set( chars );   // set StrBuf from char *

    cout << "chars[] = \"" << chars << "\"\n";
    cout << "sb.Text() returns \"" << sb.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
chars[] = "string"
sb.Text() returns "string"
```

*StrBuf::Set(const char *, int)*

Set a `StrBuf` from a string of a specified length.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const char *buf`	pointer to the first byte of the string
	`int len`	length of the string
Returns	`void`	

Notes

Initialize the `StrBuf` before calling `Set()`.

Exactly `len` bytes are copied from the string to the `StrBuf`. The `length` of the `StrBuf` is set to the `len` argument.

Any memory allocated for the `StrBuf`'s `buffer` is separate from the memory for the string.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    char chars[] = "xyzzy";
    StrBuf sb;

    sb.Set( chars, 3 );   // set StrBuf from len bytes of char *

    cout << "chars[] = \"" << chars << "\"\n";
    cout << "sb.Text() returns \"" << sb.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
chars[] = "xyzzy"
sb.Text() returns "xyz"
```

*StrBuf::Set(const StrPtr *)*

Set a `StrBuf` from a pointer to a `StrPtr`.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const StrPtr *s`	pointer to the `StrPtr` instance
Returns	`void`	

Notes

Initialize the `StrBuf` and the `StrPtr` before calling `Set()`.

Any memory allocated for the `StrBuf`'s `buffer` is separate from the memory for the `StrPtr`'s `buffer`.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrRef sr( "xyz" );
    StrPtr *sp = &sr;
    StrBuf sb;

    sb.Set( sp );   // set StrBuf from StrPtr *

    cout << sp- → Text() returns \" << sp->Text() << "\"\n";
    cout << "sb.Text() returns \"" << sb.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
sp->Text() returns "xyz"
sb.Text() returns "xyz"
```

StrBuf::Set(const StrPtr &)

Set a `StrBuf` from a reference of a `StrPtr`. Arguments are commonly instances of classes derived from the `StrPtr` class, such as `StrRef` and `StrBuf`.

Virtual?	No	
Class	`StrBuf`	
Arguments	`const StrPtr &s`	reference of the `StrPtr` instance
Returns	`void`	

Notes

Initialize the `StrBuf` and the `StrPtr` before calling `Set()`.

Any memory allocated for the `StrBuf`'s `buffer` is separate from the memory for the `StrPtr`'s `buffer`.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrRef sr;
    StrBuf sbs;
    StrBuf sbt;

    sr.Set( "xyz" );
    sbt.Set( sr );    // set StrBuf from StrRef

    cout << "sr.Text() returns \"" << sr.Text() << "\"\n";
    cout << "sbt.Text() returns \"" << sbt.Text() << "\"\n\n";

    sbs.Set( "abc" );
    sbt.Set( sbs );   // set StrBuf from StrBuf

    cout << "sbs.Text() returns \"" << sbs.Text() << "\"\n";
    cout << "sbt.Text() returns \"" << sbt.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
sr.Text() returns "xyz"
sbt.Text() returns "xyz"

sbs.Text() returns "abc"
sbt.Text() returns "abc"
```

StrBuf::StringInit()

Initialize a **`StrBuf`**.

Virtual?	No
Class	`StrBuf`
Arguments	None
Returns	`void`

Notes

`StringInit()` initializes the `StrBuf` to contain a zero-length null buffer.

Normally when a `StrBuf` is created, it is initialized using the `StrBuf` constructor. However, there may be specialized cases where memory has already been allocated for a `StrBuf` instance, but the memory was not allocated through the normal mechanisms that would result in the `StrBuf` constructor initializing the instance. For these specialized cases, `StringInit()` is appropriate for initializing a `StrBuf` instance.

After a `StrBuf` has been used, calling `StringInit()` for the instance can result in a memory leak. Specifically, once the `buffer` member has been pointed to memory other than `nullStrBuf`, calling `StringInit()` for the instance will abandon the memory.

In most cases, it is preferable to use an alternative such as one of the following:

```
sb1 = StrRef::Null();

sb2.Clear();
sb2.Terminate();

sb3.Set( "" );

sb4 = "";
```

See also

`StrBuf::Clear()` `StrBuf::Set()` `StrBuf::Terminate()` `StrBuf::operator =( char * )` `StrRef::Null()`

Example

```
#include <iostream>
#include <errno.h>

#include <stdhdrs.h>
#include <strbuf.h>
```

```
#define NSTRBUFS     5
#define CHUNKSIZE    1024
#define STRBUFSIZE   sizeof( StrBuf )

int main( int argc, char **argv )
{
    char chunk[ CHUNKSIZE ];
    int chunkFree = CHUNKSIZE;
    char *pchunkStart = &chunk[ 0 ];
    char *pchunk;

    int iStrBuf;

    // Initialize the StrBufs in the chunk.

    for ( iStrBuf = 0, pchunk = pchunkStart;
         iStrBuf < NSTRBUFS;
         iStrBuf, pchunk += STRBUFSIZE )
    {
        // Ensure that there's enough free left in the chunk for a
StrBuf.
        if ( (chunkFree -= STRBUFSIZE) &lt; 0)
        {
            cout &lt;&lt; "Not enough free left in the chunk!\n";
            return ENOMEM;
        }

        // Initialize and set the value of the StrBuf.

        ((StrBuf *)pchunk)-&gt;StringInit();
        *(StrBuf *)pchunk &lt;&lt; iStrBuf + 73;
    }

    // Print the StrBufs. Do this in a separate loop so as to provide
```

```
    // some evidence that the above loop didn't corrupt adjacent
StrBufs.
    for ( iStrBuf = 0, pchunk = pchunkStart;
          iStrBuf &lt; NSTRBUFS;
          iStrBuf, pchunk += STRBUFSIZE )
    {
        cout << "StrBuf " << iStrBuf + 1 << " contains \"";
        cout << ((StrBuf *)pchunk)->Text() << "\"\n";
    }
}
```

Executing the preceding code produces the following output:

```
StrBuf 1 contains "73"
StrBuf 2 contains "74"
StrBuf 3 contains "75"
StrBuf 4 contains "76"
StrBuf 5 contains "77"
```

StrBuf::Terminate()

Null-terminate the string pointed to by the `buffer` member of a `StrBuf`. The null byte is placed in the buffer at the location indicated by the `length` member.

Virtual?	No
Class	`StrBuf`
Arguments	None
Returns	`void`

Notes

Initialize the `StrBuf` before calling `Terminate()`.

The `length` member of the `StrBuf` is effectively unchanged by `Terminate()`.

Example

`Terminate()` is defined in `strbuf.h` as follows:

```
void Terminate()
{
```

```
    Extend( 0 ); --length;
}
```

Terminate() null-terminates the string by calling **Extend(0)**, which also increments the **length** member; the **length** is then decremented within **Terminate()**, leaving it unchanged.

See also

StrBuf::StringInit()

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf sb;

    sb.Set( "xyzzy" );

    cout << "Prior to sb.SetLength( 3 ) and sb.Terminate():\n";
    cout << "  sb.Length() returns " << sb.Length() << "\n";
    cout << "  sb.Text() returns \"" << sb.Text() << "\"\n\n";

    sb.SetLength( 3 );

    cout << "After sb.SetLength( 3 ) but prior to sb.Terminate():\n";
    cout << "  sb.Length() returns " << sb.Length() << "\n";
    cout << "  sb.Text() returns \"" << sb.Text() << "\"\n\n";

    sb.Terminate();      // null-terminate the string at length

    cout << "After sb.SetLength( 3 ) and sb.Terminate():\n";
    cout << "  sb.Length() returns " << sb.Length() << "\n";
    cout << "  sb.Text() returns \"" << sb.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
Prior to sb.SetLength( 3 ) and sb.Terminate():
  sb.Length() returns 5
  sb.Text() returns "xyzzy"

After sb.SetLength( 3 ) but prior to sb.Terminate():
  sb.Length() returns 3
  sb.Text() returns "xyzzy"

After sb.SetLength( 3 ) and sb.Terminate():
  sb.Length() returns 3
  sb.Text() returns "xyz"
```

StrDict methods

This section lists the available StrDict methods.

StrDict::GetVar(const StrPtr &)

Return the value of the specified variable, or `NULL` if not defined.

Virtual?	No	
Class	`StrDict`	
Arguments	`const StrPtr &var`	the name of the variable to look up
Returns	`StrPtr *`	the value, or `NULL` if not defined

Notes

For the most part, all of the following methods are equivalent:

- `StrDict::GetVar( const StrPtr & )`
- `StrDict::GetVar( const char * )`
- `StrDict::GetVar( const char *, Error * )`
- `StrDict::GetVar( const StrPtr &, int )`
- `StrDict::GetVar( const StrPtr &, int, int )`
- `StrDict::GetVar( int, StrPtr &, StrPtr & )`

The `var` argument must specify the name of a variable in the `StrDict` that you're trying to look up. In some instances, variables in a `StrDict` are named according to the convention `FOOx` or `FOOx,y` - one example is the tagged output of `p4 filelog`. Calling `GetVar()` with these numbers as arguments saves you the work of manually constructing the variable name by using `itoa()` and `Append()`.

The version of `GetVar()` that returns an `int` is useful for iterating through a `StrDict`; the `int` argument is an index into the `StrDict`, and the two `StrPtr` arguments are set to contain the variable and value found at that index, if any. This method returns zero if there was no variable at the specified index.

Example

The implementation of `ClientUser::OutputStat()` in `clientuser.cc` provides a good source example:

```
void ClientUser::OutputStat( StrDict *varList )
{
    int i;
    StrBuf msg;
    StrRef var, val;

    // Dump out the variables, using the GetVar( x ) interface.
    // Don't display the function, which is only relevant to rpc.
    for ( i = 0; varList->GetVar( i, var, val ); i++ )
    {
        if ( var == "func" ) continue;

        // otherAction and otherOpen go at level 2, as per 99.1 +
earlier
        msg.Clear();
        msg << var << " " << val;
        char level = strncmp( var.Text(), "other", 5 ) ? '1' : '2';
        OutputInfo( level, msg.Text() );
    }

    // blank line
    OutputInfo( '0', "" );
}
```

An example of output:

```
% p4 -Ztag filelog file.c

... depotFile //depot/depot/source/file.c
... rev0 3
... change0 1949
... action0 integrate
... type0 text
... time0 1017363022
... user0 testuser
... client0 testuser-luey
... desc0 <enter description here>
... how0,0 ignored
... file0,0 //depot/depot/source/old.c
... srev0,0 #1
... erev0,0 #2
... how0,1 ignored

...
```

StrDict::GetVar(const char *)

Return the value of the specified variable, or `NULL` if not defined.

Virtual?	No	
Class	`StrDict`	
Arguments	`const char *var`	the name of the variable to look up
Returns	`StrPtr *`	the value, or `NULL` if not defined

Notes

For the most part, all of the `GetVar()` methods are equivalent.

For details, see `StrDict::GetVar( const StrPtr & )`

StrDict::GetVar(const char *, Error *)

Return the value of the specified variable, or `NULL` if not defined.

Virtual?	No	
Class	`StrDict`	
Arguments	`const char *var`	the name of the variable to look up
	`Error *e`	an error message indicating that the required parameter `var` was not set
Returns	`StrPtr *`	the value, or `NULL` if not defined

Notes

For the most part, all of the `GetVar()` methods are equivalent.

For details, see `StrDict::GetVar( const StrPtr & )`

StrDict::GetVar(const StrPtr &, int)

Return the value of the specified variable, or `NULL` if not defined.

Virtual?	No	
Class	`StrDict`	
Arguments	`const StrPtr &var`	the name of the variable to look up
	`int x`	appended to the variable's name
Returns	`StrPtr *`	the value, or `NULL` if not defined

Notes

For the most part, all of the `GetVar()` methods are equivalent.

For details, see `StrDict::GetVar( const StrPtr & )`

StrDict::GetVar(const StrPtr &, int, int)

Return the value of the specified variable, or `NULL` if not defined.

Virtual?	No	
Class	`StrDict`	
Arguments	`const StrPtr &var`	the name of the variable to look up

	`int x`	appended to the variable's name
	`int y`	appended to the variable's name
Returns	`StrPtr *`	the value, or `NULL` if not defined

Notes

For the most part, all of the `GetVar()` methods are equivalent.

For details, see `StrDict::GetVar( const StrPtr & )`

StrDict::GetVar(int, StrPtr &, StrPtr &)

Return the value of the specified variable, or `NULL` if not defined.

Virtual?	No	
Class	`StrDict`	
Arguments	`int i`	the index of a variable in the `StrDict`
	`StrPtr &var`	the name of the variable at that index, if any
	`StrPtr &val`	the value found at that index, if any
Returns	`int`	the value, or zero if no variable found

Notes

This method is typically used when iterating through a `StrDict`.

For the most part, all of the `GetVar()` methods are equivalent.

For details, see `StrDict::GetVar( const StrPtr & )`

*StrDict::Load(FILE *)*

Unmarshals the `StrDict` from a file.

Virtual?	No	
Class	`StrDict`	
Arguments	`FILE *i`	the file to load from
Returns	`int`	always equals 1

Notes

Load() loads a **StrDict** from a file previously created by **Save()**.

Example

The following example "loads" a **StrDict** by reading it from **stdin**.

```
MyStrDict sd;
ClientUser ui;

sd.Load( stdin );
ui.OutputStat( &sd );
```

Given a marshaled **StrDict** on **stdin**, the code produces the following output:

```
> cat marshaled.strdict

depotFile=//depot/file.c
clientFile=c:\test\depot\file.c
headAction=edit
headType=text
headTime=1020067607
headRev=4
headChange=2042
headModTime 1020067484
func=client-FstatInfo

> a.out < marshaled.strdict

... depotFile //depot/file.c
... clientFile clientFile=c:\test\depot\file.c
... headAction edit
... headType text
... headTime 1020067607
... headRev 4
... headChange 2042
... headModTime 1020067484
```

StrDict::Save(FILE *)

Marshals the `StrDict` into a text file.

Virtual?	No	
Class	`StrDict`	
Arguments	`FILE *out`	the file to save to
Returns	`int`	always equals 1

Notes

`Save()` stores the `StrDict` in a marshaled form to a text file, which can be recovered by using `Load()`.

Example

The following example "saves" a `StrDict` by writing it to `stdout`.

```
void MyClientUser::OutputStat( StrDict *varList )
{
    varList->Save( stdout );
}
```

Executing the preceding code produces the following output:

```
> a.out fstat //depot/file.c

depotFile=//depot/file.c
clientFile=c:\test\depot\file.c
headAction=edit
headType=text
headTime=1020067607
headRev=4
headChange=2042
headModTime=1020067484
func=client-FstatInfo
```

StrDict::SetArgv(int, char *const *)

Set a list of values, such as the arguments to a Helix server command.

Virtual?	No	
Class	`StrDict`	
Arguments	`int argc`	the number of variables (arguments)
	`char *const *argv`	the variables (arguments) themselves
Returns	`void`	

Notes

`SetArgv()` is typically used when setting command arguments in `ClientApi`.

Example

`p4api.cc` provides an example of using `SetArgv()` to set arguments.

```
int main( int argc, char **argv )
{
    ClientUser ui;
    ClientApi client;
    Error e;

    // Any special protocol mods
    // client.SetProtocol( "tag", "" );

    // Connect to server
    client.Init( &e );

    // Run the command "argv[1] argv[2...]"
    client.SetArgv( argc - 2, argv + 2 );
    client.Run( argv[1], &ui );

    // Close connection
    client.Final( &e );

    return 0;
}
```

StrNum methods

This section lists the available StrNum methods.

StrNum::StrNum(int) (constructor)

Create a `StrNum`, either unset or with a value.

Virtual?	No	
Class	`StrNum`	
Arguments	`int v`	the number to store (optional)
Returns	`StrNum`	

Notes

A `StrNum` always stores numbers using base ten.

To create a `StrNum` without a value, call `StrNum()` without an argument.

Example

The following example creates a `StrNum` and displays it:

```
#include <iostream>
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrNum sn = StrNum( 1666 );
    cout << "sn.Text() returns \"" << sn.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
sn.Text() returns "1666"
```

StrNum::Set(int)

Set a `StrNum`'s value.

Virtual?	No	
Class	`StrNum`	
Arguments	`int v`	the number to store
Returns	`void`	

Notes

A `StrNum` always stores numbers using base ten.

Example

```
#include <iostream>
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrNum sn;
    sn.Set ( 1666 );
    cout << "sn.Text() returns \"" << sn.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
sn.Text() returns "1666"
```

StrOps methods

This section lists the available StrOps methods.

StrOps::Caps(StrBuf &)

Convert the first character in a string (in place) to uppercase.

Virtual?	No	
Class	`StrOps`	
Arguments	`StrBuf &o`	the string to capitalize
Returns	`void`	

Example

```
#include <stdhdrs.h>
#include <strbuf.h>
#include <strops.h>

int main( int argc, char **argv )
{
    StrBuf sb;

    sb.Set( "xyzzy" );
    printf( "Before: %s\n", sb.Text() );

    StrOps::Caps( sb );
    printf( "After: %s\n", sb.Text() );

    return 0;
}
```

Executing the preceding code produces the following output:

```
Before: xyzzy
After: Xyzzy
```

StrOps::Dump(const StrPtr &)

Pretty-print a string to `stdout`

Virtual?	No	
Class	`StrOps`	
Arguments	`StrPtr &o`	the string to dump
Returns	`void`	

Notes

Unprintable characters are displayed as hexadecimal ASCII values, surrounded by greater-than/less-than characters.

Example

```
#include <stdhdrs.h>
#include <strbuf.h>
#include <strops.h>

int main( int argc, char **argv )
{
    StrBuf sb;
    sb.Set( "\tXyzzy" );

    StrOps::Dump( sb );

    return 0;
}
```

Executing the preceding code produces the following output:

```
<09>Xyzzy
```

StrOps::Expand(StrBuf &, StrPtr &, StrDict &)

Expand "`%*var*%" strings into corresponding "`*val*" strings from a `StrDict`.

Virtual?	No	
Class	`StrOps`	
Arguments	`StrBuf &o`	the output string
	`StrPtr &s`	the input string
	`StrDict &d`	the var/value pairs to look up
Returns	`void`	

Notes

This function provides a way to quickly expand variables from a `StrDict` into a `StrBuf`.

Example

This small program demonstrates the `Expand()` method in an `OutputStat()` implementation:

```
void MyClientUser::OutputStat( StrDict *varList )
{
    StrBuf s = StrBuf();
    s.Set( "File: %depotFile%  Rev: %rev%" );
    StrBuf o = StrBuf();
    StrOps::Expand( o, s, *varList );
    StrOps::Dump( o );
}

int main( int argc, char **argv )
{
    ClientApi client;
    MyClientUser ui;
    Error e;

    client.SetProtocol( "tag", "" );
    client.Init( &e );
    client.SetArgv( 1, ++argv );
    client.Run( "files", &ui );
    return client.Final( &e );
}
```

Executing the preceding code produces the following output:

```
% a.out *
File: //depot/src/file1.c  Rev: 4
File: //depot/src/file2.c  Rev: 2
```

StrOps::Expand2(StrBuf &, StrPtr &, StrDict &)

Expand "[%*var*%|*alt*\]" strings into corresponding "*val*" strings from a `StrDict`, or "*alt*" if "*var*" is undefined.

Virtual?	No	
Class	`StrOps`	
Arguments	`StrBuf &o`	the output string

	`StrPtr &s`	the input string
	`StrDict &d`	the var/value pairs to look up
Returns	`void`	

Notes

Like `Expand()`, this function provides a way to quickly expand variables from a `StrDict` into a `StrBuf`, with the additional feature of providing alternate text if the value is not defined.

The exact syntax of the expression to be expanded is:

`[ text1 %var% text2 | alt \]`

If variable "*var*" has value "val" in the `StrDict d`, the expression expands to:

`text1 val text2`

otherwise, it expands to:

`alt`

See the example for details.

Example

This small program demonstrates the `Expand2()` method in an `OutputStat()` implementation:

```
void MyClientUser::OutputStat( StrDict *varList )
{
    StrBuf s = StrBuf();
    s.Set( "stat: [File: %depotFile%|No file]!" );

    StrBuf o = StrBuf();
    StrOps::Expand2( o, s, *varList );

    StrOps::Dump( o );
}

int main( int argc, char **argv )
{
    ClientApi client;
    MyClientUser ui;
    Error e;
```

```
    client.SetProtocol( "tag", "" );
    client.Init( &e );

    client.SetArgv( argc - 2, argv + 2 );
    client.Run( argv[1], &ui );

    return client.Final( &e );
}
```

Executing the preceding code produces the following output:

```
% a.out files *
stat: File: //depot/src/file1.c!
stat: File: //depot/src/file2.c!

% a.out labels
stat: No file!
stat: No file!
stat: No file!
stat: No file!
stat: No file!
```

StrOps::Indent(StrBuf &, const StrPtr &)

Make a copy of a string, with each line indented.

Virtual?	No	
Class	`StrOps`	
Arguments	`StrBuf &o`	the output string
	`StrPtr &s`	the input string
Returns	`void`	

Notes

This function reads the input string `s` and copies it to the output string `o`, with each line indented with a single tab.

Example

```
StrBuf s = StrBuf();
s.Set( "abc\ndef\nghi\n" );

StrBuf o = StrBuf();
StrOps::Indent( o, s );

printf( "Before:\n%s", s.Text() );
printf( "After:\n%s", o.Text() );
```

Executing the preceding code produces the following output:

```
Before:
abc
def
ghi
After:
        abc
        def
        ghi
```

*StrOps::Lines(StrBuf &, char *[], int)*

Break a string apart at line breaks.

Virtual?	No	
Class	`StrOps`	
Arguments	`StrBuf &o`	the input string
	`char *vec[]`	the output array
	`int maxVec`	the maximum number of lines to handle
Returns	`int`	the actual number of lines handled

Notes

This function handles all types of line breaks: “\r”, “\n”, and “\r\n”.

Example

```
StrBuf o = StrBuf();
o.Set( "abc\ndef\nghi\n" );

printf( "Input StrBuf:\n%s\n", o.Text() );

char *vec[4];
int l = StrOps::Lines( o, vec, 4 );

for ( ; l ; l-- )
{
    printf( "Line %d: %s\n", l, vec[l-1] );
}
```

Executing the preceding code produces the following output:

```
Input StrBuf:
abc
def
ghi

Line 3: abc
Line 2: def
Line 1: ghi
```

StrOps::Lower(StrBuf &)

Convert each character in a string (in place) to lowercase

Virtual?	No	
Class	`StrOps`	
Arguments	`StrBuf &o`	the string to convert to lowercase
Returns	`void`	

Notes

This function modifies an original string in place by converting all uppercase characters to lowercase.

Example

```
StrBuf o = StrBuf();
o.Set( "xYzZy" );

printf( "Before: %s\n", o );
StrOps::Lower( o );
printf( "After:  %s\n", o );

return 0;
```

Executing the preceding code produces the following output:

```
% a.out
Before: xYzZy
After:  xyzzy
```

*StrOps::OtoX(const unsigned char *, int, StrBuf &)*

Convert an octet stream into hex.

Virtual?	No	
Class	`StrOps`	
Arguments	`char *octet`	the input stream
	`int len`	length of the input in bytes
	`StrBuf &x`	the output string
Returns	`void`	

Notes

This function converts the input stream into a string of hexadecimal numbers, with each byte from the input being represented as exactly two hex digits.

Example

```
const unsigned char stream[3] = { 'f', 'o', 'o' };
StrBuf hex;
StrOps::OtoX( stream, 3, hex );
```

```
StrOps::Dump( hex );
return 0;
```

Executing the preceding code produces the following output:

```
% a.out
666F6F
```

StrOps::Replace(StrBuf &, const StrPtr &, const StrPtr &, const StrPtr &)

Replace substrings in a `StrPtr` and store the result to a `StrBuf`.

Virtual?	No	
Class	`StrOps`	
Arguments	`StrBuf &o`	the output string
	`StrPtr &i`	the input string
	`StrBuf &s`	the substring to match
	`StrPtr &r`	the substring to replace `s`
Returns	`void`	

Notes

This function reads the input string `i` and copies it to the output string `o`, after replacing each occurrence of the string `s` with string `r`.

Example

```
StrBuf i = StrBuf();
i.Set( "PerForce is PerForce, of course, of course!" );

StrBuf wrong, right;
wrong.Set( "PerForce" );
right.Set( "Perforce" );

StrBuf o = StrBuf();

StrOps::Replace( o, i, wrong, right );
```

```
StrOps::Dump( o );
```

Executing the preceding code produces the following output:

```
% a.out
Perforce is Perforce, of course, of course!
```

StrOps::Sub(StrPtr &, char, char)

Substitute instances of one character for another.

Virtual?	No	
Class	`StrOps`	
Arguments	`StrPtr &string`	the string on which to operate
	`target`	the target character
	`replace`	the character with which to replace `target`
Returns	`void`	

Notes

This function substitutes the `replace` character for every instance of the `target` character in the input `string`. The substitution is performed in place.

Example

```
#include <stdhdrs.h>
#include <strbuf.h>
#include <strops.h>

int main( int argc, char **argv )
{
    StrBuf sb;
    sb.Set( "\tPassword" );

    StrOps::Sub( sb, 'o', '0' );
    StrOps::Sub( sb, 'a', '4' );
```

```
    StrOps::Dump( sb );

    return 0;
}
```

Executing the preceding code produces the following output:

```
P4ssw0rd
```

StrOps::Upper(StrBuf &)

Convert each character in a string (in place) to uppercase

Virtual?	No	
Class	`StrOps`	
Arguments	`StrBuf &o`	the string to convert to uppercase
Returns	`void`	

Notes

This function modifies an original string in place by converting all lowercase characters to uppercase.

Example

```
StrBuf o = StrBuf();
o.Set( "xYzZy" );

printf( "Before: %s\n", o );
StrOps::Upper( o );
printf( "After:  %s\n", o );

return 0;
```

Executing the preceding code produces the following output:

```
% a.out
Before: xYzZy
After:  XYZZY
```

StrOps::Words(StrBuf &, const char *[], char *[], int)

Break a string apart at whitespace.

Virtual?	No	
Class	`StrOps`	
Arguments	`StrBuf &tmp`	a temporary string
	`const char *buf`	the input string
	`char *vec[]`	the output array
	`int maxVec`	the maximum number of words to handle
Returns	`int`	the actual number of words handled

Notes

This function uses the `isAspace()` function to define whitespace.

Example

```
StrBuf o = StrBuf();
StrBuf tmp = StrBuf();
o.Set( "abc\tdef   ghi\nxyz xyzzy plugh" );

printf( "Input StrBuf:\n%s\n", o.Text() );

char *vec[5];
int w = StrOps::Words( tmp, o, vec, 5 );

for ( ; w ; w-- )
{
    printf( "Word %d: %s\n", w, vec[w-1] );
}

return 0;
```

Executing the preceding code produces the following output:

```
Input StrBuf:
abc     def   ghi
```

```
xyz xyzzy plugh

Word 5: xyzzy
Word 4: xyz
Word 3: ghi
Word 2: def
Word 1: abc
```

StrOps::XtoO(char *, unsigned char *, int)

Convert a hex string into an octet stream.

Virtual?	No	
Class	`StrOps`	
Arguments	`char *x`	the input hex string
	`char *octet`	the output stream
	`int octlen`	the length of the output, in bytes
Returns	`void`	

Notes

This function converts the input hexadecimal string into the stream of bytes that it represents.

Example

```
char *hex = "666F6F";
unsigned char oct[4];

StrOps::XtoO( hex, oct, 3 );
oct[3] = '\0';

printf( "%s", oct );

return 0;
```

Executing the preceding code produces the following output:

```
% a.out
foo
```

StrPtr methods

This section lists the available StrPtr methods.

StrPtr::Atoi()

Return the numeric value, if any, represented by this `StrPtr`'s buffer.

Virtual?	No	
Class	`StrPtr`	
Arguments	None	
Returns	`int`	integer value of the string

Notes

`StrPtr::Atoi()` is equivalent to calling `atoi(StrPtr::Text())`. Non-numeric strings typically return a value of zero.

Example

```
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;
    StrBuf str2;

    str1.Set( "123" );
    str2.Set( "234" );

    printf( "%s + %s = %d\n",
        str1.Text(), str2.Text(), str1.Atoi() + str2.Atoi() );
}
```

Executing the preceding code produces the following output:

```
123 + 234 = 357
```

StrPtr::CCompare(const StrPtr &)

Case insensitive comparison of two `StrPtr`s.

Virtual?	No	
Class	`StrPtr`	
Arguments	`const StrPtr &s`	the `StrPtr` to compare this one with
Returns	`int`	zero if identical, nonzero if different

Notes

`StrPtr::CCompare()` is a wrapper for `stricmp()` or `strcasecmp()`. Its return value, if nonzero, indicates which of the two strings is "greater" in the ASCII sense.

See also

`StrPtr::XCompare()` `StrPtr::Compare()`

Example

```
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1, str2, str3;

    str1.Set( "abc" );
    str2.Set( "Abc" );
    str3.Set( "xyz" );

    if ( str1.CCompare( str2 ) == 0 )
        printf( "%s == %s\n", str1.Text(), str2.Text() );
    else
        printf( "%s != %s\n", str1.Text(), str2.Text() );
```

```
    if ( str1.CCompare( str3) == 0 )
        printf( "%s == %s\n", str1.Text(), str3.Text() );
    else
        printf( "%s != %s\n", str1.Text(), str3.Text() );

    return 0;
}
```

Executing the preceding code produces the following output:

```
abc == Abc
abc != xyz
```

StrPtr::Compare(const StrPtr &)

Comparison of two `StrPtr`s, with case sensitivity based on client platform.

Virtual?	No	
Class	`StrPtr`	
Arguments	`const StrPtr &s`	the `StrPtr` to compare this one with
Returns	`int`	zero if identical, nonzero if different

Notes

`StrPtr::Compare()` is a wrapper for `zstrcmp()`. Its return value, if nonzero, indicates which of the two strings is "greater" in the ASCII sense.

See also

`StrPtr::CCompare()` `StrPtr::XCompare()`.

Example

```
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1, str2, str3;
    str1.Set( "abc" );
```

```
    str2.Set( "Abc" );
    str3.Set( "xyz" );

    if ( str1.Compare( str2 ) == 0 )
        printf( "%s == %s\n", str1.Text(), str2.Text() );
    else
        printf( "%s != %s\n", str1.Text(), str2.Text() );

    if ( str1.Compare( str3 ) == 0 )
        printf( "%s == %s\n", str1.Text(), str3.Text() );
    else
        printf( "%s != %s\n", str1.Text(), str3.Text() );

    return 0;
}
```

Executing the preceding code produces the following output on Windows:

```
abc == Abc
abc != xyz
```

and on Unix

```
abc != Abc
abc != xyz
```

StrPtr::Contains(const StrPtr &)

Look for a substring and, if found, return it.

Virtual?	No	
Class	`StrPtr`	
Arguments	`const StrPtr &s`	the substring to look for
Returns	`char *`	the start of the substring if found, otherwise `NULL`

Notes

`StrPtr::Contains()` returns a pointer to the `StrPtr's` buffer, rather than allocating a new buffer for the substring. If it cannot find the substring, `Contains()` returns `NULL`.

Example

```
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1, str2;

    str1.Set( "the quick brown fox jumps over the lazy dog" );
    str2.Set( "brown fox" );

    printf( "%s\n", str1.Contains( str2 ) );

    return 0;
}
```

Executing the preceding code produces the following output:

```
brown fox jumps over the lazy dog
```

StrPtr::Length()

Return the length of this `StrPtr`.

Virtual?	No	
Class	`StrPtr`	
Arguments	None	
Returns	`int`	the length of this `StrPtr`

Example

```
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;
```

```
    str1.Set( "This string" );

    printf( "%s is %d bytes long\n", str1, str1.Length() );
    return 0;
}
```

Executing the preceding code produces the following output:

```
This string is 11 bytes long
```

StrPtr::operator [](int)

Return the character at the specified index.

Virtual?	No	
Class	`StrPtr`	
Arguments	`int x`	the index to look in
Returns	`char`	the character at that index

Notes

This operator does no bounds checking, and can therefore return data from beyond the end of the string.

Example

```
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;

    str1.Set( "the quick brown fox jumps over the lazy dog" );

    printf( "%c%c%c%c%c\n", str1[1], str1[2], str1[35], str1[35], str1
[12] );
```

```
    return 0;
}
```

Executing the preceding code produces the following output:

```
hello
```

StrPtr::operators ==, !=, >, <, <=, >= (const char *)

Case-sensitive comparison operators between `StrPtr` and `char *`.

Virtual?	No	
Class	`StrPtr`	
Arguments	`const char *buf`	the string to compare with
Returns	`int`	zero if the comparison is false, nonzero if true.

Notes

These operators are typically used in simple comparisons between `StrPtr`s, such as to see whether two `StrPtr`s contain the same string, or whether one is greater than the other, ASCII-wise. The comparison is always case-sensitive.

Example

```
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;

    str1.Set( "This string" );

    printf( "%s", str1.Text());
    if ( str1 == "that string" ) printf( " == " );
    if ( str1 > "that string" )  printf( " > " );
    if ( str1 < "that string" )  printf( " < " );
    printf( "that string" );
```

```
    return 0;
}
```

Executing the preceding code produces the following output:

```
This string < that string
```

(Note that “t” > “T” in ASCII.)

StrPtr::operators ==, !=, >, <, <=, >= (const StrPtr &)

Case-sensitive comparison operators between `StrPtr` and `StrPtr`.

Virtual?	No	
Class	`StrPtr`	
Arguments	`const StrPtr & buf`	the string to compare with
Returns	`int`	zero if the comparison is false, nonzero if true.

Notes

These operators are typically used in simple comparisons between `StrPtr`s, such as to see whether two `StrPtr`s contain the same string, or whether one is greater than the other, ASCII-wise. The comparison is always case-sensitive.

Example

```
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1, str2;

    str1.Set( "This string" );
    str2.Set( "that string" );

    printf( "%s", str1.Text());
    if ( str1 == str2 ) printf( " == " );
    if ( str1 > str2 )  printf( " > " );
    if ( str1 < str2 )  printf( " < " );
```

```
    printf( "%s\n", str2.Text() );
    return 0;
}
```

Executing the preceding code produces the following output:

```
This string < that string
```

(Note that “t” > “T” in ASCII.)

StrPtr::Text()

Return the `char *` containing this `StrPtr`'s text.

Virtual?	No	
Class	`StrPtr`	
Arguments	None	
Returns	`char *`	This `StrPtr`'s buffer

Notes

`StrPtr::Text()` and `StrPtr::Value()` are exactly equivalent. Their most typical use is converting a `StrPtr` to a `char *` for functions outside of the client API to use.

Example

```
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;

    str1.Set( "the quick brown fox jumps over the lazy dog" );

    printf( "%s\n", str1.Text() );

    return 0;
}
```

Executing the preceding code produces the following output:

```
the quick brown fox jumps over the lazy dog
```

StrPtr::Value()

Return the `char *` containing this `StrPtr`'s text.

Virtual?	No	
Class	`StrPtr`	
Arguments	None	
Returns	`char *`	This `StrPtr`'s buffer

Notes

`StrPtr::Value()` is the deprecated form of `StrPtr::Text()`. The two functions are equivalent. Their most typical use is converting a `StrPtr` to a `char *` for functions outside of the client API to use.

Example

```
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;

    str1.Set( "the quick brown fox jumps over the lazy dog" );

    printf( "%s\n", str1.Value() );

    return 0;
}
```

Executing the preceding code produces the following output:

```
the quick brown fox jumps over the lazy dog
```

StrPtr::XCompare(const StrPtr &)

Case sensitive comparison of two `StrPtr`s.

Virtual?	No	
Class	`StrPtr`	
Arguments	`const StrPtr &s`	the `StrPtr` to compare this one with
Returns	`int`	zero if identical, nonzero if different

Notes

`StrPtr::XCompare()` is a wrapper for `strcmp()`. Its return value, if nonzero, indicates which of the two strings is "greater" in the ASCII sense.

See also

`StrPtr::CCompare()` `StrPtr::Compare()`

Example

```
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1, str2, str3;

    str1.Set( "abc" );
    str2.Set( "Abc" );
    str3.Set( "xyz" );

    if ( str1.XCompare( str2 ) == 0 )
        printf( "%s == %s\n", str1.Text(), str2.Text() );
    else
        printf( "%s != %s\n", str1.Text(), str2.Text() );

    if ( str1.XCompare( str3 ) == 0 )
        printf( "%s == %s\n", str1.Text(), str3.Text() );
    else
        printf( "%s != %s\n", str1.Text(), str3.Text() );
```

```
    return 0;
}
```

Executing the preceding code produces the following output:

```
abc != Abc
abc != xyz
```

StrRef methods

This section lists the available StrRef methods.

StrRef::StrRef() (constructor)

Construct a `StrRef`, and leave it unset.

Virtual?	No
Class	`StrRef`
Arguments	None

Notes

If arguments are provided, the constructor calls `Set()` with them.

StrRef::StrRef(const StrPtr &) (constructor)

Construct a `StrRef`, referencing an existing string.

Virtual?	No	
Class	`StrRef`	
Arguments	`const StrPtr &`	a `StrPtr` to reference
Returns	`StrRef`	

Notes

If arguments are provided, the constructor calls `Set()` with them.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;

    str1.Set( "abc" );
    StrRef sr = StrRef( str1 );

    cout << "str1 = \"" << str1.Text() << "\"\n";
    cout << "sr.Text() returns \"" << sr.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
str1 = "abc"
sr.Text() returns "abc"
```

*StrRef::StrRef(const char *) (constructor)*

Construct a `StrRef`, referencing an existing string.

Virtual?	No	
Class	`StrRef`	
Arguments	`char *buf`	a null-terminated string to reference
Returns	`StrRef`	

Notes

If arguments are provided, the constructor calls `Set()` with them.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    char chars[] = "abc";
    StrRef sr = StrRef( chars );

    cout << "chars[] = \"" << chars << "\"\n";
    cout << "sr.Text() returns \"" << sr.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
chars[] = "abc"
sr.Text() returns "abc"
```

*StrRef::StrRef(const char * , int) (constructor)*

Construct a `StrRef`, referencing an existing string.

Virtual?	No	
Class	`StrRef`	
Arguments	`char *buf`	a null-terminated string to reference
	`int len`	the string length
Returns	`StrRef`	

Notes

If arguments are provided, the constructor calls `Set()` with them.

`StrRef::Set()` does not copy the target string; it simply creates a pointer to it. Be sure that the `StrRef` pointing to the target string does not outlive the target string.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    char chars[] = "xyzzy";
    StrRef sr = StrRef( chars, 3 );
    StrBuf sb;
    sb.Set( sr );

    printf( "chars[] = \"%s\"\n", chars );
    printf( "sr.Text() returns \"%s\"\n", sr.Text() );
    printf( "sb.Text() returns \"%s\"\n", sb.Text() );

    return 0;
}
```

Executing the preceding code produces the following output:

```
chars[] = "xyzzy"
sr.Text() returns "xyzzy"
sb.Text() returns "xyz"
```

StrRef::Null()

Return a null `StrPtr`.

Virtual?	No	
Class	`StrRef`	
Arguments	None	
Returns	`StrPtr`	an empty `StrPtr`

Notes

`StrRef::Null()` is a static function.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;

    str1.Set( "abc" );
    StrRef sr = StrRef( str1 );

    if ( sr == StrRef::Null() )
        cout << "str1 was null\n";
    else
        cout << "str1 was not null\n";
}
```

Executing the preceding code produces the following output:

```
str1 was not null
```

StrRef::operator =(StrPtr &)

Set a `StrPtr` to reference an existing `StrPtr` or null-terminated string.

Virtual?	No	
Class	`StrRef`	
Arguments	`StrPtr &s`	the `StrPtr` to reference
Returns	`void`	

Notes

The `=` operator is equivalent to calling `Set()`.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;

    str1.Set( "xyz" );
    StrRef sr = str1;

    cout << "str1 = \"" << str1.Text() << "\"\n";
    cout << "sr.Text() returns \"" << sr.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
str1 = "xyz"
sr.Text() returns "xyz"
```

*StrRef::operator =(char *)*

Set a `StrPtr` to reference an existing `StrPtr` or null-terminated string.

Virtual?	No	
Class	`StrRef`	
Arguments	`char *buf`	the null-terminated string to reference.
Returns	`void`	

Notes

The `=` operator is equivalent to calling `Set()`.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    char chars[] = "xyz";
    StrRef sr;

    sr = chars;

    cout << "chars[] = \"" << chars << "\"\n";
    cout << "sr.Text() returns \"" << sr.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
chars[] = "xyz"
sr.Text() returns "xyz"
```

StrRef::operator +=(int)

Increase a `StrRef's` pointer and decrease its length.

Virtual?	No	
Class	`StrRef`	
Arguments	`int len`	the amount by which to move the pointer
Returns	`void`	

Notes

This method has the effect of removing `len` characters from the beginning of the `StrRef`. It does not, however, free the memory allocated to those characters.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    char chars[] = "xyzzy";
    StrRef sr = StrRef( chars );

    sr += 3;

    cout << "chars[] = \"" << chars << "\"\n";
    cout << "sr.Text() returns \"" << sr.Text() << "\"\n";
}
```

Executing the preceding code produces the following output:

```
chars[] = "xyzzy"
sr.Text() returns "zy"
```

*StrRef::Set(char *)*

Set a `StrRef` to reference an existing null-terminated string.

Virtual?	No	
Class	`StrRef`	
Arguments	`char *buf`	the null-terminated string to reference
Returns	`void`	

Notes

`StrRef::Set()` does not copy the target string; it simply establishes a pointer to it. Be sure that the `StrRef` pointing to the target string does not outlive the target string.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    char chars[] = "xyz";
    StrRef sr;

    sr.Set( chars );

    cout << "chars[] = \"" << chars << "\"\n";
    cout << "sr.Text() returns \"" << sr.Text() << "'\"n";
}
```

Executing the preceding code produces the following output:

```
chars[] = "xyz"
sr.Text() returns "xyz"
```

*StrRef::Set(char * , int)*

Set a `StrRef` to reference an existing null-terminated string.

Virtual?	No	
Class	`StrRef`	
Arguments	`char *buf`	the null-terminated string to reference
	`int len`	the length of the string
Returns	`void`	

Notes

`StrRef::Set()` does not copy the target string; it simply establishes a pointer to it. Be sure that the `StrRef` pointing to the target string does not outlive the target string.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    char chars[] = "xyzzy";
    StrBuf sb;
    StrRef sr;
    sb.Set( chars );
    sr.Set( chars, 3 );

    printf( "chars[] = \"%s\"\n", chars );
    printf( "sr.Text() returns \"%s\"\n", sr.Text() );
    printf( "sb.Text() returns \"%s\"\n", sb.Text() );

    return 0;
}
```

Executing the preceding code produces the following output:

```
chars[] = "xyzzy"
sr.Text() returns "xyzzy"
sb.Text() returns "xyz"
```

*StrRef::Set(const StrPtr *)*

Set a `StrRef` to reference an existing `StrPtr`.

Virtual?	No	
Class	`StrRef`	
Arguments	`const StrPtr *s`	the value to set
Returns	`void`	

Notes

`StrRef::Set()` does not copy the target string; it simply establishes a pointer to it. Be sure that the `StrRef` pointing to the target string does not outlive the target string.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrRef sr;
    sr.Set( "xyz" );

    cout << "sr.Text() returns \"" << sr.Text() << "'\"n";
}
```

Executing the preceding code produces the following output:

```
sr.Text() returns "xyz"
```

StrRef::Set(const StrPtr &)

Set a `StrRef` to reference an existing `StrPtr`.

Virtual?	No	
Class	`StrRef`	
Arguments	`const StrPtr &s`	the `StrPtr` to reference
Returns	`void`	

Notes

`StrRef::Set()` does not copy the target string; it simply establishes a pointer to it. Be sure that the `StrRef` pointing to the target string does not outlive the target string.

Example

```
#include <iostream>

#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;
    StrRef sr;

    str1.Set ( "xyz" );
    sr.Set( str1 );

    cout << "str1 = \"" << str1.Text() << "\"\n";
    cout << "sr.Text() returns \"" << sr.Text() << "'\"n";
}
```

Executing the preceding code produces the following output:

```
str1 = "xyz"
sr.Text() returns "xyz"
```

Glossary

A

access level

A permission assigned to a user to control which commands the user can execute. See also the 'protections' entry in this glossary and the 'p4 protect' command in the P4 Command Reference.

admin access

An access level that gives the user permission to privileged commands, usually super privileges.

APC

The Alternative PHP Cache, a free, open, and robust framework for caching and optimizing PHP intermediate code.

archive

1. For replication, versioned files (as opposed to database metadata). 2. For the 'p4 archive' command, a special depot in which to copy the server data (versioned files and metadata).

atomic change transaction

Grouping operations affecting a number of files in a single transaction. If all operations in the transaction succeed, all the files are updated. If any operation in the transaction fails, none of the files are updated.

avatar

A visual representation of a Swarm user or group. Avatars are used in Swarm to show involvement in or ownership of projects, groups, changelists, reviews, comments, etc. See also the "Gravatar" entry in this glossary.

B

base

For files: The file revision that contains the most common edits or changes among the file revisions in the source file and target file paths. For checked out streams: The public have version from which the checked out version is derived.

binary file type

A Helix server file type assigned to a non-text file. By default, the contents of each revision are stored in full, and file revision is stored in compressed format.

branch

(noun) A set of related files that exist at a specific location in the Perforce depot as a result of being copied to that location, as opposed to being added to that location. A group of related files is often referred to as a codeline. (verb) To create a codeline by copying another codeline with the 'p4 integrate', 'p4 copy', or 'p4 populate' command.

branch form

The form that appears when you use the 'p4 branch' command to create or modify a branch specification.

branch mapping

Specifies how a branch is to be created or integrated by defining the location, the files, and the exclusions of the original codeline and the target codeline. The branch mapping is used by the integration process to create and update branches.

branch view

A specification of the branching relationship between two codelines in the depot. Each branch view has a unique name and defines how files are mapped from the originating codeline to the target codeline. This is the same as branch mapping.

broker

Helix Broker, a server process that intercepts commands to the Helix server and is able to run scripts on the commands before sending them to the Helix server.

C

change review

The process of sending email to users who have registered their interest in changelists that include specified files in the depot.

changelist

A list of files, their version numbers, the changes made to the files, and a description of the changes made. A changelist is the basic unit of versioned work in Helix server. The changes specified in the changelist are not stored in the depot until the changelist is submitted to the depot. See also atomic change transaction and changelist number.

changelist form

The form that appears when you modify a changelist using the 'p4 change' command.

changelist number

An integer that identifies a changelist. Submitted changelist numbers are ordinal (increasing), but not necessarily consecutive. For example, 103, 105, 108, 109. A pending changelist number might be assigned a different value upon submission.

check in

To submit a file to the Helix server depot.

check out

To designate one or more files, or a stream, for edit.

checkpoint

A backup copy of the underlying metadata at a particular moment in time. A checkpoint can recreate db.user, db.protect, and other db.* files. See also metadata.

classic depot

A repository of Helix server files that is not streams-based. Uses the Perforce file revision model, not the graph model. The default depot name is depot. See also default depot, stream depot, and graph depot.

client form

The form you use to define a client workspace, such as with the 'p4 client' or 'p4 workspace' commands.

client name

A name that uniquely identifies the current client workspace. Client workspaces, labels, and branch specifications cannot share the same name.

client root

The topmost (root) directory of a client workspace. If two or more client workspaces are located on one machine, they should not share a client root directory.

client side

The right-hand side of a mapping within a client view, specifying where the corresponding depot files are located in the client workspace.

client workspace

Directories on your machine where you work on file revisions that are managed by Helix server. By default, this name is set to the name of the machine on which your client workspace is located, but it can be overridden. Client workspaces, labels, and branch specifications cannot share the same name.

code review

A process in Helix Swarm by which other developers can see your code, provide feedback, and approve or reject your changes.

codeline

A set of files that evolve collectively. One codeline can be branched from another, allowing each set of files to evolve separately.

comment

Feedback provided in Helix Swarm on a changelist, review, job, or a file within a changelist or review.

commit server

A server that is part of an edge/commit system that processes submitted files (checkins), global workspaces, and promoted shelves.

conflict

1. A situation where two users open the same file for edit. One user submits the file, after which the other user cannot submit unless the file is resolved. 2. A resolve where the same line is changed when merging one file into another. This type of conflict occurs when the comparison of two files to a base yields different results, indicating that the files have been changed in different ways. In this case, the merge cannot be done automatically and must be resolved manually. See file conflict.

copy up

A Helix server best practice to copy (and not merge) changes from less stable lines to more stable lines. See also merge.

counter

A numeric variable used to track variables such as changelists, checkpoints, and reviews.

CSRF

Cross-Site Request Forgery, a form of web-based attack that exploits the trust that a site has in a user's web browser.

D

default changelist

The changelist used by a file add, edit, or delete, unless a numbered changelist is specified. A default pending changelist is created automatically when a file is opened for edit.

deleted file

In Helix server, a file with its head revision marked as deleted. Older revisions of the file are still available. in Helix server, a deleted file is simply another revision of the file.

delta

The differences between two files.

depot

A file repository hosted on the server. A depot is the top-level unit of storage for versioned files (depot files or source files) within a Helix Core server. It contains all versions of all files ever submitted to the depot. There can be multiple depots on a single installation.

depot root

The topmost (root) directory for a depot.

depot side

The left side of any client view mapping, specifying the location of files in a depot.

depot syntax

Helix server syntax for specifying the location of files in the depot. Depot syntax begins with: //depot/

diff

(noun) A set of lines that do not match when two files, or stream versions, are compared. A conflict is a pair of unequal diffs between each of two files and a base, or between two versions of a stream. (verb) To compare the contents of files or file revisions, or of stream versions. See also conflict.

donor file

The file from which changes are taken when propagating changes from one file to another.

E

edge server

A replica server that is part of an edge/commit system that is able to process most read/write commands, including 'p4 integrate', and also deliver versioned files (depot files).

exclusionary access

A permission that denies access to the specified files.

exclusionary mapping

A view mapping that excludes specific files or directories.

extension

Similar to a trigger, but more modern. See "Helix Core Server Administrator Guide" on "Extensions".

F

file conflict

In a three-way file merge, a situation in which two revisions of a file differ from each other and from their base file. Also, an attempt to submit a file that is not an edit of the head revision of the file in the depot, which typically occurs when another user opens the file for edit after you have opened the file for edit.

file pattern

Helix server command line syntax that enables you to specify files using wildcards.

file repository

The master copy of all files, which is shared by all users. In Helix server, this is called the depot.

file revision

A specific version of a file within the depot. Each revision is assigned a number, in sequence. Any revision can be accessed in the depot by its revision number, preceded by a pound sign (#), for example testfile#3.

file tree

All the subdirectories and files under a given root directory.

file type

An attribute that determines how Helix server stores and diffs a particular file. Examples of file types are text and binary.

fix

A job that has been closed in a changelist.

form

A screen displayed by certain Helix server commands. For example, you use the change form to enter comments about a particular changelist to verify the affected files.

forwarding replica

A replica server that can process read-only commands and deliver versioned files (depot files). One or more replicate servers can significantly improve performance by offloading some of the master server load. In many cases, a forwarding replica can become a disaster recovery server.

G

Git Fusion

A Perforce product that integrates Git with Helix, offering enterprise-ready Git repository management, and workflows that allow Git and Helix server users to collaborate on the same projects using their preferred tools.

graph depot

A depot of type graph that is used to store Git repos in the Helix server. See also Helix4Git and classic depot.

group

A feature in Helix server that makes it easier to manage permissions for multiple users.

H

have list

The list of file revisions currently in the client workspace.

head revision

The most recent revision of a file within the depot. Because file revisions are numbered sequentially, this revision is the highest-numbered revision of that file.

heartbeat

A process that allows one server to monitor another server, such as a standby server monitoring the master server (see the p4 heartbeat command).

Helix server

The Helix server depot and metadata; also, the program that manages the depot and metadata, also called Helix Core server.

Helix TeamHub

A Perforce management platform for code and artifact repository. TeamHub offers built-in support for Git, SVN, Mercurial, Maven, and more.

Helix4Git

Perforce solution for teams using Git. Helix4Git offers both speed and scalability and supports hybrid environments consisting of Git repositories and 'classic' Helix server depots.

hybrid workspace

A workspace that maps to files stored in a depot of the classic Perforce file revision model as well as to files stored in a repo of the graph model associated with git.

I

iconv

A PHP extension that performs character set conversion, and is an interface to the GNU libiconv library.

integrate

To compare two sets of files (for example, two codeline branches) and determine which changes in one set apply to the other, determine if the changes have already been propagated, and propagate any outstanding changes from one set to another.

J

job

A user-defined unit of work tracked by Helix server. The job template determines what information is tracked. The template can be modified by the Helix server system administrator. A job describes work to be done, such as a bug fix. Associating a job with a changelist records which changes fixed the bug.

job daemon

A program that checks the Helix server machine daily to determine if any jobs are open. If so, the daemon sends an email message to interested users, informing them the number of jobs in each category, the severity of each job, and more.

job specification

A form describing the fields and possible values for each job stored in the Helix server machine.

job view

A syntax used for searching Helix server jobs.

journal

A file containing a record of every change made to the Helix server's metadata since the time of the last checkpoint. This file grows as each Helix server transaction is logged. The file should be automatically truncated and renamed into a numbered journal when a checkpoint is taken.

journal rotation

The process of renaming the current journal to a numbered journal file.

journaling

The process of recording changes made to the Helix server's metadata.

L

label

A named list of user-specified file revisions.

label view

The view that specifies which filenames in the depot can be stored in a particular label.

lazy copy

A method used by Helix server to make internal copies of files without duplicating file content in the depot. A lazy copy points to the original versioned file (depot file). Lazy copies minimize the consumption of disk space by storing references to the original file instead of copies of the file.

license file

A file that ensures that the number of Helix server users on your site does not exceed the number for which you have paid.

list access

A protection level that enables you to run reporting commands but prevents access to the contents of files.

local depot

Any depot located on the currently specified Helix server.

local syntax

The syntax for specifying a filename that is specific to an operating system.

lock

1. A file lock that prevents other clients from submitting the locked file. Files are unlocked with the 'p4 unlock' command or by submitting the changelist that contains the locked file. 2. A database lock that prevents another process from modifying the database db.* file.

log

Error output from the Helix server. To specify a log file, set the P4LOG environment variable or use the p4d -L flag when starting the service.

M

mapping

A single line in a view, consisting of a left side and a right side that specify the correspondences between files in the depot and files in a client, label, or branch. See also workspace view, branch view, and label view.

MDS checksum

The method used by Helix server to verify the integrity of versioned files (depot files).

merge

1. To create new files from existing files, preserving their ancestry (branching). 2. To propagate changes from one set of files to another. 3. The process of combining the contents of two conflicting file revisions into a single file, typically using a merge tool like P4Merge.

merge file

A file generated by the Helix server from two conflicting file revisions.

metadata

The data stored by the Helix server that describes the files in the depot, the current state of client workspaces, protections, users, labels, and branches. Metadata is stored in the Perforce database and is separate from the archive files that users submit.

modification time or modtime

The time a file was last changed.

MPM

Multi-Processing Module, a component of the Apache web server that is responsible for binding to network ports, accepting requests, and dispatch operations to handle the request.

N

nonexistent revision

A completely empty revision of any file. Syncing to a nonexistent revision of a file removes it from your workspace. An empty file revision created by deleting a file and the #none revision specifier are examples of nonexistent file revisions.

numbered changelist

A pending changelist to which Helix server has assigned a number.

O

opened file

A file you have checked out in your client workspace as a result of a Helix Core server operation (such as an edit, add, delete, integrate). Opening a file from your operating system file browser is not tracked by Helix Core server.

owner

The Helix server user who created a particular client, branch, or label.

P

p4

1. The Helix Core server command line program. 2. The command you issue to execute commands from the operating system command line.

p4d

The program that runs the Helix server; p4d manages depot files and metadata.

P4PHP

The PHP interface to the Helix API, which enables you to write PHP code that interacts with a Helix server machine.

PECL

PHP Extension Community Library, a library of extensions that can be added to PHP to improve and extend its functionality.

pending changelist

A changelist that has not been submitted.

Perforce

Perforce Software, Inc., a leading provider of enterprise-scale software solutions to technology developers and development operations (“DevOps”) teams requiring productivity, visibility, and scale during all phases of the development lifecycle.

project

In Helix Swarm, a group of Helix server users who are working together on a specific codebase, defined by one or more branches of code, along with options for a job filter, automated test integration, and automated deployment.

protections

The permissions stored in the Helix server’s protections table.

proxy server

A Helix server that stores versioned files. A proxy server does not perform any commands. It serves versioned files to Helix server clients.

R

RCS format

Revision Control System format. Used for storing revisions of text files in versioned files (depot files). RCS format uses reverse delta encoding for file storage. Helix server uses RCS format to store text files. See also reverse delta storage.

read access

A protection level that enables you to read the contents of files managed by Helix server but not make any changes.

remote depot

A depot located on another Helix server accessed by the current Helix server.

replica

A Helix server that contains a full or partial copy of metadata from a master Helix server. Replica servers are typically updated every second to stay synchronized with the master server.

repo

A graph depot contains one or more repos, and each repo contains files from Git users.

reresolve

The process of resolving a file after the file is resolved and before it is submitted.

resolve

The process you use to manage the differences between two revisions of a file, or two versions of a stream. You can choose to resolve file conflicts by selecting the source or target file to be submitted, by merging the contents of conflicting files, or by making additional changes. To resolve stream conflicts, you can choose to accept the public source, accept the checked out target, manually accept changes, or combine path fields of the public and checked out version while accepting all other changes made in the checked out version.

reverse delta storage

The method that Helix server uses to store revisions of text files. Helix server stores the changes between each revision and its previous revision, plus the full text of the head revision.

revert

To discard the changes you have made to a file in the client workspace before a submit.

review access

A special protections level that includes read and list accesses and grants permission to run the p4 review command.

review daemon

A program that periodically checks the Helix server machine to determine if any changelists have been submitted. If so, the daemon sends an email message to users who have subscribed to any of the files

included in those changelists, informing them of changes in files they are interested in.

revision number

A number indicating which revision of the file is being referred to, typically designated with a pound sign (#).

revision range

A range of revision numbers for a specified file, specified as the low and high end of the range. For example, myfile#5,7 specifies revisions 5 through 7 of myfile.

revision specification

A suffix to a filename that specifies a particular revision of that file. Revision specifiers can be revision numbers, a revision range, change numbers, label names, date/time specifications, or client names.

RPM

RPM Package Manager. A tool, and package format, for managing the installation, updates, and removal of software packages for Linux distributions such as Red Hat Enterprise Linux, the Fedora Project, and the CentOS Project.

S

server data

The combination of server metadata (the Helix server database) and the depot files (your organization's versioned source code and binary assets).

server root

The topmost directory in which p4d stores its metadata (db.* files) and all versioned files (depot files or source files). To specify the server root, set the P4ROOT environment variable or use the p4d -r flag.

service

In the Helix Core server, the shared versioning service that responds to requests from Helix server client applications. The Helix server (p4d) maintains depot files and metadata describing the files and also tracks the state of client workspaces.

shelve

The process of temporarily storing files in the Helix server without checking in a changelist.

status

For a changelist, a value that indicates whether the changelist is new, pending, or submitted. For a job, a value that indicates whether the job is open, closed, or suspended. You can customize job statuses. For the 'p4 status' command, by default the files opened and the files that need to be reconciled.

storage record

An entry within the db.storage table to track references to an archive file.

stream

A "branch" with built-in rules that determines what changes should be propagated and in what order they should be propagated.

stream depot

A depot used with streams and stream clients. Has structured branching, unlike the free-form branching of a "classic" depot. Uses the Perforce file revision model, not the graph model. See also classic depot and graph depot.

stream hierarchy

The set of parent-to-child relationships between streams in a stream depot.

submit

To send a pending changelist into the Helix server depot for processing.

super access

An access level that gives the user permission to run every Helix server command, including commands that set protections, install triggers, or shut down the service for maintenance.

symlink file type

A Helix server file type assigned to symbolic links. On platforms that do not support symbolic links, symlink files appear as small text files.

sync

To copy a file revision (or set of file revisions) from the Helix server depot to a client workspace.

T

target file

The file that receives the changes from the donor file when you integrate changes between two codelines.

text file type

Helix server file type assigned to a file that contains only ASCII text, including Unicode text. See also binary file type.

theirs

The revision in the depot with which the client file (your file) is merged when you resolve a file conflict. When you are working with branched files, theirs is the donor file.

three-way merge

The process of combining three file revisions. During a three-way merge, you can identify where conflicting changes have occurred and specify how you want to resolve the conflicts.

trigger

A script that is automatically invoked by Helix server when various conditions are met. (See "Helix Core Server Administrator Guide" on "Triggers".)

two-way merge

The process of combining two file revisions. In a two-way merge, you can see differences between the files.

typemap

A table in Helix server in which you assign file types to files.

U

user

The identifier that Helix server uses to determine who is performing an operation. The three types of users are standard, service, and operator.

V

versioned file

Source files stored in the Helix server depot, including one or more revisions. Also known as an archive file. Versioned files typically use the naming convention 'filenamev' or '1.changelist.gz'.

view

A description of the relationship between two sets of files. See workspace view, label view, branch view.

W

wildcard

A special character used to match other characters in strings. The following wildcards are available in Helix server: * matches anything except a slash; ... matches anything including slashes; %%0 through %%9 is used for parameter substitution in views.

workspace

See client workspace.

workspace view

A set of mappings that specifies the correspondence between file locations in the depot and the client workspace.

write access

A protection level that enables you to run commands that alter the contents of files in the depot. Write access includes read and list accesses.

X

XSS

Cross-Site Scripting, a form of web-based attack that injects malicious code into a user's web browser.

Y

yours

The edited version of a file in your client workspace when you resolve a file. Also, the target file when you integrate a branched file.

License Statements

To get a listing of the third-party software licenses that Helix Core server uses, at the command line, type the `p4 help legal` command.

To get a listing of the third-party software licenses that the local client (such as P4V) uses, at the command line, type the `p4 help -l legal` command.