Perforce 2004.2
C/C++ API User's Guide

September 2004

Table of Contents

Preface

About This Manual

This is the *Perforce 2004.2 C/C++ API User's Guide*.

This guide contains details about using the Perforce client API to create client programs that interact correctly with the Perforce server. Be sure to read the code in the API's header and C++ files in conjunction with this guide.

Interfaces for Perl, Ruby, Python, and other languages are available from our website:

```
http://www.perforce.com/perforce/loadsupp.html#api
```

Please give us feedback

If you have any feedback for us, or detect any errors in this guide, please email details to `manual@perforce.com`.

Chapter 1 Overview

Release compatibility of the API

The Perforce Client 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, clients that you create using the API can run against previous releases of the Perforce server and will probably run against later releases of the Perforce server.

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

Purpose of the API

The Perforce Client API enables you to create client programs that interact with end users, send commands to a Perforce server and process data returned from the server. The API is a programmatic interface, and does not send commands directly to the server.

Architecture of the API

The basic client session is managed by a C++ class called `ClientApi`. All user interaction with the client is channeled through the `ClientUser` C++ class. The default methods of `ClientUser` implement the `p4` command line interface. To create custom client programs, create subclasses based on `ClientUser`.

API files

The Perforce client API consists of header files, link libraries, and the reference implementation of the `ClientUser` class. Only the libraries are platform-specific.

The API is packaged as an archive or zip file. The source code for the libraries is proprietary and is not included. To download the API, go to the Perforce FTP site and download the file for your platform. For example, to obtain the Macintosh version using a Web browser, use the following URL:
`ftp://ftp.perforce.com/perforce/r01.1/bin.mac/` and download `p4api.sit.hqx`.

(Specific API files can vary from release to release, and so are not individually described here.)

Chapter 2 Client Programming

Compiling and linking client programs

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

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

Link order

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

1. `libclient.a`
2. `librpc.a`
3. `libsupp.a`

In the Windows distribution, these files are named `libclient.lib`, `librpc.lib`, and `libsupp.lib` respectively.

Compiler support

UNIX

For all UNIX platforms, you can use the `gcc` compiler to compile client programs with the Perforce Client API. On Solaris, you can also use the Forte compiler.

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. For example, Solaris requires the following compiler flags:

```
-lsocket -lnsl.
```

The gcc 2.95.2 compiler on Solaris 7 requires the following flag:

```
-Dconst_char=const char
```

Windows NT

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`
- `wsock32.lib`
- `advapi32.lib`

Macintosh

To create an MPW tool, link with the following libraries:

- `Interfacelib`
- `PPCToolLibs.o`
- `PLStringFuncsPPC.lib`
- `MSL-MPWCRuntime.lib`
- `MSL-C.PPC-MPW(NL).Lib`
- `MSL-C++.PPC.Lib`
- `ThreadsLib`
- `Mathlib`
- `InternetConfigLib`
- `OpenTransportLib`
- `OpenTptInternetLib`
- `OpenTransportAppPPC.o`

The compiler option `Enums Always Int` must be on.

VMS

Link with `sys$library:libcxxstd.olb/lib`.

Sample Jamfile

The following example shows a Jamfile that can be used to build `samplemain.cc`, a Perforce client program. (The example that the API is installed in the `api` subdirectory.)

```
C++FLAGS = -g -D_GNU_SOURCE ;
LINK = c++ ;
OPTIM = ;
Main samplemain : samplemain.cc ;
ObjectHdrs samplemain : api ;
LinkLibraries samplemain : api/libclient.a api/librpc.a api/libsupp.a ;
```

For more about `jam`, see “Building with Jam” on page 16.

Sample Makefile

The following is a `gnumake` file for building `samplemain.cc`, a Perforce client program. (The example that the API is installed in the `api` subdirectory.)

```
SOURCES = samplemain.cc
INCLUDES = -Iapi
OBJECTS = ${SOURCES:.cc=.o}
LIBRARIES = api/libclient.a api/librpc.a api/libsupp.a
BINARY = samplemain

C++ = c++
C++FLAGS = -c -g -D_GNU_SOURCE
LINK = c++
LINKFLAGS =

.cc.o :
   ${C++} ${C++FLAGS} $< ${INCLUDES}

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

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

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. Jam documentation, source code, and links to precompiled binaries are available from the Jam product information page at:

```
http://www.perforce.com/jam/jam.html
```

The P4API distribution contains the necessary header files (`*.h`) and libraries (`libclient.a`, `librpc.a`, `libsupp.a`) required to compile and link a client application. The distribution also includes a sample client program in C++, `samplemain.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 client program `samplemain.cc` is a portable, minimal client application, which we can use as an example. For purposes of this example, assume a Linux system.

Compile and link `samplemain.cc` as follows:

```
$ cc  -c -o samplemain.o -D_GNU_SOURCE -O2 -DOS_LINUX -DOS_LINUX24 \
> -DOS_LINUXX86 -DOS_LINUX24X86 -I. -Imsgs -Isupport -Isys samplemain.cc
$ gcc -o samplemain samplemain.o libclient.a librpc.a libsupp.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 `samplemain` 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++ samplemain.o
Link samplemain
Chmod1 samplemain
...updated 2 target(s)...

$ samplemain 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: 2002/09/24 12:15:39 PDT
Server version: P4D/LINUX22X86/2002.1/32489 (2002/04/12)
Server license: Your Company 10 users (expires 2003/02/10)
```

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++ samplemain.o
  Link samplemain
  Chmod1 samplemain
  ...updated 2 target(s)...
$  cat build_sample
cc -c -o samplemain.o -O2 -DOS_LINUX -DOS_LINUX42 -DOS_LINUXX86 \
-DOS_LINUX42X86 -I. -Imsgs -Isupport -Isys samplemain.cc
gcc -o samplemain samplemain.o libclient.a librpc.a libsupp.a
chmod 711 samplemain
```

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 `samplemain info` or most other Perforce commands:

```
$ samplemain changes -m 1
Change 372 on 2002/09/23 by you@you:home:sunflower 'Building API'
```

As you can see, `samplemain` is a usable full-featured command line Perforce 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 server

Client programs interact with the Perforce server by:

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

The Perforce server does not maintain any kind of session identifier. The server identifies the sender of commands by its combination of Perforce user name and client specification name. Different processes that use the same combination of user and client name are not distinguished by the Perforce 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.

Perforce settings on the client machine

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

To override client computer settings, your client program can call `Set` methods.

Client computer settings take precedence as follows, highest to lowest:

1. Values set within a Perforce 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 Perforce software or gathered from the system

Connecting to the server

To connect to the Perforce 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 Perforce 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 page 21.

Displaying Perforce forms

Perforce 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 Perforce 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` client program, rather than during a single invocation. (For details about the `-o` and `-i` global options, see the *Command Reference*.)

All form-related commands can return a form descriptor. Your client program 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 client program must pass a `StrDict` object to `ClientUser::OutputStat()`. Your client program can override the `OutputStat()` method in a class derived from `ClientUser`. The Perforce Client 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

The Perforce server (release 99.2 and higher) 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 Perforce server to return tagged data, your client program must call `SetProtocol("tag", "")` before connecting to the server. To extract values from tagged data, use the `GetVars` method.

The following Perforce commands can return tagged output. A release number, when present, indicates the first Perforce server release that supports tagged output for the command.

p4 branch -o	p4 filelog	p4 labels
p4 branches	p4 fixes (2000.1)	p4 logger (2000.2)
p4 changes	p4 group -o	p4 opened
p4 client -o	p4 groups (2004.2)	p4 protect -o
p4 clients	p4 info (2003.2)	p4 trigger -o
p4 counters (2000.2)	p4 job -o	p4 typemap -o (2000.1)
p4 describe	p4 jobs	p4 user -o
p4 diff2 (2004.2)	p4 jobspec -o	p4 users
p4 dirs	p4 label -o	p4 where (2004.2)

The tagged output of some commands may have changed since the first release in this table. For details, see the release notes at:

`http://www.perforce.com/perforce/doc.042/user/p4apinotes.txt`

To obtain output in the form used by earlier revisions of Perforce, set the `api` variable according to the notes for `SetProtocol()`.

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 client program is finished interacting with the Perforce server, it must disconnect as illustrated below:

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

To ensure the client program can exit successfully, make sure your client program calls `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 "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 Perforce 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, Perforce servers invoked `OutputError()` directly with formatted error text.

Class overviews

The following classes comprise the Perforce API. Public methods for these classes are documented in “Public Methods Reference” on page 33.

ClientApi - Perforce server connections and commands

The `ClientApi` class represents a connection with the Perforce 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 Perforce 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 Perforce 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.

ClientUser - I/O for Perforce 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 Perforce client 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 Perforce 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 - Perforce 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.

KeepAlive - support for client-side disconnection

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

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.

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 Perforce 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.

Chapter 3 Public Methods Reference

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::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 `samplemain.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 Perforce 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 in order 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 `samplemain.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 Perforce 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::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()`, 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	`const StrPtr& -` a reference to the hostname

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::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::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 `samplemain.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 *)

Run a Perforce command and return when it completes.

Virtual?	No
Class	`ClientApi`
Arguments	`const char* func` - the name of the command to run
Returns	`void`

Notes

The `func` argument to `Run()` is the Perforce 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.

In contrast to `RunTag()`, `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.

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::RunTag(const char *, ClientUser *)

Queue a Perforce command without waiting for it to complete.

Virtual?	No
Class	`ClientApi`
Arguments	`const char* func` - the command to run `ClientUser* ui` - the `ClientUser` object to use for I/O
Returns	`void`

Notes

`RunTag()` can be used to queue as many as four calls to the server at once, with the results distinguished only by the individual `ClientUser` objects passed to each `RunTag()` call. When four calls are placed on the queue, `RunTag()` starts to execute them by calling `WaitTag()` to dispatch them to the server and get output back.

The benefit of using `RunTag()` and `WaitTag()` rather than simply `RunTag()` is that the performance penalty of network latency is reduced. Rather than sending one command out and waiting until the results return, one command can be sent while output from an earlier one is coming in. This approach reduces the time consumed by network latency by about half. The commands are always executed in the order that they were placed on the queue with `RunTag()`.

Using `RunTag()` to queue multiple commands is known to have trouble with large requests and password authentication. The limit of four "tags" might change in future releases. If in doubt, use `Run()`.

Use `RunTag()` to wait for commands in the queue to complete without being nudged off by additional `RunTag()` calls.

See Also

`ClientApi::WaitTag()`

Example

The following example demonstrates five Perforce commands queued to run against the server asynchronously using `RunTag()` and `WaitTag()`.

After the fourth call to `RunTag()` completes, the command from the first call is nudged off the queue and forced to complete; after the fifth call, the second command completes.

The final call to `RunTag()` forces the three remaining commands to complete.

It is good practice to make checks with `WaitTag()` between `RunTag()` calls to ensure that the connection is still up, but they have been left out of this example for brevity's sake.

```
int main( int argc, char **argv )
   ClientUser ui1;
   ClientUser ui2;
   ClientUser ui3;
   ClientUser ui4;
   ClientUser ui5;
   ClientApi client;
   Error e;

   client.Init( &e );

   client.RunTag("depots", &ui1);
   printf("\n1\n");
   client.RunTag("depots", &ui2);
   printf("\n2\n");
   client.RunTag("depots", &ui3);
   printf("\n3\n");
   client.RunTag("depots", &ui4);
   printf("\n4\n");
   client.RunTag("depots", &ui5);
   printf("\n5\n");
   client.WaitTag();
   printf("\n6\n");

   client.Final( &e );

   return 0;
}
> a.out

1

2

3
Depot depot 2001/04/05 local subdir depot/... 'Created by user. '

4
Depot depot 2001/04/05 local subdir depot/... 'Created by user. '

5
Depot depot 2001/04/05 local subdir depot/... 'Created by user. '
Depot depot 2001/04/05 local subdir depot/... 'Created by user. '
Depot depot 2001/04/05 local subdir depot/... 'Created by user. '

6
```

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 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 );
```

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::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	`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;
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::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	`void`

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 example demonstrates how to hard-code a password into a client program without making it 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	`void`

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 example demonstrates how to hard-code a password into a client program without making it 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 = "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()` after calling `Init()` and before calling `Run()`.

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()` after calling `Init()` and before calling `Run()`.

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.

To protect your code against changes in server level, use the `api` variable.

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 may break (that is, start producing tagged output) when running against a 2003.2 or newer server. To prevent this from happening, 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 greater.

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 Perforce API or connecting to newer Perforce 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 Perforce 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::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::WaitTag(ClientUser *)

Waits for pending commands to complete.

Virtual?	No
Class	`ClientApi`
Arguments	`ClientUser*` - the `ClientUser` corresponding to the newest command to terminate (default = null)
Returns	`void`

Notes

`WaitTag()` waits for commands initiated with `RunTag()` to complete. If `WaitTag()` is called with a valid `ClientUser*` argument, it waits for each command in the queue to return, up to the command specified by the argument. If called with no argument, or an invalid `ClientUser` pointer, `WaitTag()` waits for all pending commands to complete.

Note that if even if `WaitTag()` is called with a `ClientUser` argument, commands are still always executed in the order in which they were placed on the queue with `RunTag()`.

See Also

`ClientApi::RunTag()`

Example

The following example demonstrates the use of `WaitTag()` without any arguments; the client application waits for both `info` commands to complete.

```
ClientApi client;
ClientUser ui1;
ClientUser ui2;

client.RunTag("info", &ui1);
client.RunTag("info", &ui2);
client.WaitTag();
```

ClientUser methods

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 Perforce client 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 a client program 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 a client program 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::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 Perforce 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 Perforce 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 client program 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::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 Perforce merge program; it merely invokes external merge programs (including P4WinMerge 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 client program 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 Perforce commands. Earlier versions of the Perforce 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 client program 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 Perforce. 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 Perforce 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 Perforce 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 `StrBuf`s, 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 Perforce 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.

Example

Consider the following output from `p4 fstat`:

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

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 Perforce 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

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

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 `samplemain.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

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
`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 Perforce 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 );
   }
}
```

KeepAlive methods

KeepAlive::IsAlive( )

The only method of the `KeepAlive` class, `IsAlive()` is used in client programs 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 );
```

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->Test() )
   {
       StrBuf msg;
       e->Fmt( &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->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
Returns	`StrPtr *`

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 Perforce 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 `-a` *`value`* or `-a`*`value`*. Although an argument of the form `-a` *`value`* 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 `-a` *`value`* or `-a`*`value`*; using a period allows only the `-a`*`value`* 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.
`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 < 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()`

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 Perforce 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

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
Returns	N/A

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
Returns	N/A

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) < 0 )
        {
            cout << "Not enough free left in the chunk!\n";
            return ENOMEM;
        }

        // Initialize and set the value of the StrBuf.

        ((StrBuf *)pchunk)->StringInit();
        *(StrBuf *)pchunk << 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 < 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

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 `FOO`*`x`* or `FOO`*`x`*`,`*`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
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
```

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 marshalled 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
func=client-FstatInfo
```

StrDict::SetArgv(int, char *const *)

Set a list of values, such as the arguments to a Perforce 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

`samplemain.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

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

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::Words(StrBuf &, char *[], int)

Break a string apart at whitespace.

Virtual?	No
Class	`StrOps`
Arguments	`StrBuf &o` - 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();
o.Set( "abc\tdef   ghi\nxyz xyzzy plugh" );

printf( "Input StrBuf:\n%s\n", o.Text() );

char *vec[5];
int w = StrOps::Words( 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

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()` and `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(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(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(str1.Text());
   if (str1 == str2)  printf(" == ");
   if (str1 > str2) printf(" > ");
   if (str1 < str2) printf(" < ");
   printf(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(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(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

StrRef::StrRef( ) (constructor)

Construct a `StrRef`, and leave it unset.

Virtual?	No
Class	`StrRef`
Arguments	None
Returns	`StrRef`

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"
```