Table of Contents

About This Manual ........................................................................................................... xvii

Please give us feedback ................................................................................................... xvii

Chapter 1  Overview .......................................................................................................... 1

Release compatibility of the API ..................................................................................... 1
Purpose of the API ........................................................................................................... 1
Architecture of the API .................................................................................................. 1
API files ....................................................................................................................... 1

Chapter 2  Application Programming ............................................................................. 3

Compiling and linking Perforce applications ................................................................. 3
  Link order .................................................................................................................. 3
SSL support .................................................................................................................. 3
  OpenSSL Library Version ......................................................................................... 3
  Link order for SSL support ..................................................................................... 3
Compiler support ......................................................................................................... 4
  UNIX ....................................................................................................................... 4
  Linux ...................................................................................................................... 4
  Windows .................................................................................................................. 4
  Macintosh ................................................................................................................. 5
  VMS ....................................................................................................................... 5
Sample Jamfile ............................................................................................................. 5
Sample Makefile ......................................................................................................... 5

Building with Jam .......................................................................................................... 6
  Building the sample application ............................................................................. 7
Sending commands to the versioning service ............................................................... 8
  Perforce settings on the user's machine ................................................................. 8
  Connecting to the server ....................................................................................... 9
  Displaying Perforce forms ................................................................................. 9
  Sending commands .............................................................................................. 10
  Processing data from the server ..................................................................... 10
    Tagged data ....................................................................................................... 10
    Untagged Data ................................................................................................. 11
  Disconnecting from the server ......................................................................... 12
Performing file I/O ........................................................................................................ 12

Handling errors ............................................................................................................ 15
  Connection errors ............................................................................................... 15
  Server errors ....................................................................................................... 15

Class overviews ............................................................................................................ 15
  ClientApi - Perforce server connections and commands ................................ 15
  ClientProgress - progress indicators for Perforce commands ....................... 16
  ClientUser - I/O for Perforce commands ........................................................... 16
  Error - collect and report layered errors ......................................................... 17
  ErrorLog - output error messages ................................................................... 17
Chapter 3  Public Methods Reference ................................................................. 21

ClientApi methods ................................................................. 21
ClientApi::DefineClient( const char *, Error * ) ........................................ 21
  Notes ................................................................. 21
  Example ............................................................. 21
ClientApi::DefineHost( const char *, Error * ) ........................................... 22
  Notes ................................................................. 22
  Example ............................................................. 22
ClientApi::DefineIgnoreFile( const char *, Error * ) .................................... 23
  Notes ................................................................. 23
  See also ........................................................... 23
  Example ............................................................. 23
ClientApi::DefinePassword( const char *, Error * ) ...................................... 24
  Notes ................................................................. 24
  Example ............................................................. 24
ClientApi::DefinePort( const char *, Error * ) ............................................... 25
  Notes ................................................................. 25
  Example ............................................................. 25
ClientApi::DefineUser( const char *, Error * ) ............................................... 26
  Notes ................................................................. 26
  Example ............................................................. 26
ClientApi::Dropped() ........................................................................ 27
  Notes ................................................................. 27
  Example ............................................................. 27
ClientApi::Final( Error * ) ....................................................................... 28
  Notes ................................................................. 28
  Example ............................................................. 28
ClientApi::GetClient() .......................................................................... 29
  Notes ................................................................. 29
  Example ............................................................. 29
ClientApi::GetConfig() .......................................................................... 30
  Notes ................................................................. 30
  Example ............................................................. 30
ClientApi::GetCwd() ............................................................................. 31
  Notes ................................................................. 31
  Example ............................................................. 31
ClientApi::GetHost() ............................................................................. 32
  Notes ................................................................. 32
Example ..................................................................................................... 32
ClientApi::GetIgnore() ............................................................................... 33
Notes ......................................................................................................... 33
See also .................................................................................................... 33
Example .................................................................................................. 33
ClientApi::GetIgnoreFile() ........................................................................ 34
Notes ......................................................................................................... 34
See also .................................................................................................... 34
Example .................................................................................................. 34
ClientApi::GetOs() .................................................................................... 35
Notes ......................................................................................................... 35
Example .................................................................................................... 35
ClientApi::GetPassword() ......................................................................... 36
Notes ......................................................................................................... 36
Example .................................................................................................... 36
ClientApi::GetPort() .................................................................................. 37
Notes ......................................................................................................... 37
Example .................................................................................................... 37
ClientApi::GetProtocol( const char * ) .................................................... 38
Notes ......................................................................................................... 38
Example .................................................................................................... 38
ClientApi::GetUser() ................................................................................ 39
Notes ......................................................................................................... 39
Example .................................................................................................... 39
ClientApi::Init( Error * ) ......................................................................... 40
Notes ......................................................................................................... 40
Example .................................................................................................... 40
ClientApi::Run( const char *, ClientUser * ) ............................................ 41
Notes ......................................................................................................... 41
Example .................................................................................................... 41
ClientApi::SetBreak( KeepAlive *breakCallback ) .................................... 42
Notes ......................................................................................................... 42
See also .................................................................................................... 42
Example .................................................................................................... 42
ClientApi::SetClient( const StrPtr * ) ..................................................... 44
Notes ......................................................................................................... 44
Example .................................................................................................... 44
ClientApi::SetClient( const char * ) ....................................................... 45
Notes ......................................................................................................... 45
Example .................................................................................................... 45
ClientApi::SetCwd( const StrPtr * ) ......................................................... 46
Notes ......................................................................................................... 46
Example .................................................................................................... 46
ClientApi::SetCwd( const char * ) ........................................................... 47
Notes ......................................................................................................... 47
Example .................................................................................................... 47
ClientApi::SetCwdNoReload( const StrPtr * ) ........................................... 48
Notes ......................................................................................................... 48
Example .................................................................................................... 48
ClientApi::SetCwdNoReload( const char * ) .............................................. 49
Notes ......................................................................................................... 49
Example .................................................................................................... 49
ClientApi::SetHost( const StrPtr * ) ......................................................... 50
Notes ......................................................................................................... 50
Example ................................................................................................. 50
ClientApi::SetHost( const char * ) .............................................................. 51
Notes ........................................................................................................ 51
Example ................................................................................................. 51
ClientApi::SetIgnoreFile( const StrPtr * ) .................................................. 52
Notes ........................................................................................................ 52
See also ................................................................................................. 52
Example ................................................................................................. 52
ClientApi::SetIgnoreFile( const char * ) .................................................... 53
Notes ........................................................................................................ 53
See also ................................................................................................. 53
Example ................................................................................................. 53
ClientApi::SetPassword( const StrPtr * ) ................................................... 54
Notes ........................................................................................................ 54
Example ................................................................................................. 54
ClientApi::SetPassword( const char * ) .................................................... 55
Notes ........................................................................................................ 55
Example ................................................................................................. 55
ClientApi::SetPort( const StrPtr * ) .......................................................... 56
Notes ........................................................................................................ 56
Example ................................................................................................. 56
ClientApi::SetPort( const char * ) ............................................................ 57
Notes ........................................................................................................ 57
Example ................................................................................................. 57
ClientApi::SetProg( const StrPtr * ) ......................................................... 58
Notes ........................................................................................................ 58
See also ................................................................................................. 58
Example ................................................................................................. 58
ClientApi::SetProg( const char * ) ........................................................... 59
Notes ........................................................................................................ 59
See also ................................................................................................. 59
Example ................................................................................................. 59
ClientApi::SetProtocol( char *, char * ) .................................................... 60
Notes ........................................................................................................ 60
Example ................................................................................................. 60
ClientApi::SetProtocolV( char * ) ............................................................. 62
Notes ........................................................................................................ 62
Example ................................................................................................. 62
ClientApi::SetTicketFile( const StrPtr * ) .................................................. 63
Notes ........................................................................................................ 63
Example ................................................................................................. 63
ClientApi::SetTicketFile( const char * ) .................................................... 64
Notes ........................................................................................................ 64
Example ................................................................................................. 64
ClientApi::SetUi( ClientUser * ) ................................................................. 65
Notes ........................................................................................................ 65
Example ................................................................................................. 65
ClientApi::SetUser( const StrPtr * ) .......................................................... 66
Notes ........................................................................................................ 66
Example ................................................................................................. 66
ClientApi::SetUser( const char * ) ............................................................ 67
Notes ........................................................................................................ 67
Example ................................................................................................. 67
ClientApi::SetVersion( const StrPtr * ) ..................................................... 68
<table>
<thead>
<tr>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes ...........................................</td>
<td>68</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>68</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>68</td>
</tr>
<tr>
<td>ClientApi::SetVersion( const char * )  ....</td>
<td>69</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>69</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>69</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>69</td>
</tr>
<tr>
<td>C/C++ API User’s Guide ........................</td>
<td></td>
</tr>
<tr>
<td>Notes ...........................................</td>
<td>70</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>70</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>70</td>
</tr>
<tr>
<td>ClientProgress methods .......................</td>
<td>70</td>
</tr>
<tr>
<td>ClientProgress::Description( const StrPtr *, int )</td>
<td>70</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>70</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>70</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>70</td>
</tr>
<tr>
<td>ClientProgress::Done( int ) ..................</td>
<td>71</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>71</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>71</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>71</td>
</tr>
<tr>
<td>ClientProgress::Total( long ) ...............</td>
<td>72</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>72</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>72</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>72</td>
</tr>
<tr>
<td>ClientProgress::Update( long ) .............</td>
<td>73</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>73</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>73</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>73</td>
</tr>
<tr>
<td>ClientUser methods ............................</td>
<td>74</td>
</tr>
<tr>
<td>ClientUser::CreateProgress( int ) ..........</td>
<td>74</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>74</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>74</td>
</tr>
<tr>
<td>ClientUser::Diff( FileSys *, FileSys *, int, char *, Error * )</td>
<td>75</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>75</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>75</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>75</td>
</tr>
<tr>
<td>ClientUser::Diff( FileSys *, FileSys *, FileSys *, int, char *, Error * )</td>
<td>77</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>77</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>77</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>77</td>
</tr>
<tr>
<td>ClientUser::Edit( FileSys *, Error * ) .....</td>
<td>78</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>78</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>78</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>78</td>
</tr>
<tr>
<td>ClientUser::ErrorPause( char *, Error * )</td>
<td>79</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>79</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>79</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>79</td>
</tr>
<tr>
<td>ClientUser::File( FileSysType ) ............</td>
<td>80</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>80</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>80</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>80</td>
</tr>
<tr>
<td>ClientUser::Finished() .......................</td>
<td>81</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>81</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>81</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>81</td>
</tr>
<tr>
<td>ClientUser::HandleError( Error * ) ..........</td>
<td>82</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>82</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>82</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>82</td>
</tr>
<tr>
<td>ClientUser::Help( const char *const * ) ....</td>
<td>83</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>83</td>
</tr>
<tr>
<td>See also ......................................</td>
<td>83</td>
</tr>
<tr>
<td>Example .......................................</td>
<td>83</td>
</tr>
<tr>
<td>ClientUser::InputData( StrBuf *, Error * )</td>
<td>84</td>
</tr>
<tr>
<td>Notes ..........................................</td>
<td>84</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>Error::IsWarning()</td>
<td>105</td>
</tr>
<tr>
<td>Notes</td>
<td>105</td>
</tr>
<tr>
<td>Example</td>
<td>105</td>
</tr>
<tr>
<td>Error::Net(const char *, const char *)</td>
<td>106</td>
</tr>
<tr>
<td>Notes</td>
<td>106</td>
</tr>
<tr>
<td>Example</td>
<td>106</td>
</tr>
<tr>
<td>Error::operator &lt;&lt; (int)</td>
<td>107</td>
</tr>
<tr>
<td>Notes</td>
<td>107</td>
</tr>
<tr>
<td>Example</td>
<td>107</td>
</tr>
<tr>
<td>Error::operator &lt;&lt; (char *)</td>
<td>108</td>
</tr>
<tr>
<td>Notes</td>
<td>108</td>
</tr>
<tr>
<td>Example</td>
<td>108</td>
</tr>
<tr>
<td>Error::operator &lt;&lt; (const StrPtr &amp;)</td>
<td>109</td>
</tr>
<tr>
<td>Notes</td>
<td>109</td>
</tr>
<tr>
<td>Error::operator = (Error &amp;)</td>
<td>110</td>
</tr>
<tr>
<td>Notes</td>
<td>110</td>
</tr>
<tr>
<td>Example</td>
<td>110</td>
</tr>
<tr>
<td>Error::Set(enum ErrorSeverity, const char *)</td>
<td>111</td>
</tr>
<tr>
<td>Notes</td>
<td>111</td>
</tr>
<tr>
<td>Example</td>
<td>111</td>
</tr>
<tr>
<td>Error::Set(ErrorCode &amp;)</td>
<td>112</td>
</tr>
<tr>
<td>Notes</td>
<td>112</td>
</tr>
<tr>
<td>Error::Sys(const char *, const char *)</td>
<td>113</td>
</tr>
<tr>
<td>Notes</td>
<td>113</td>
</tr>
<tr>
<td>Example</td>
<td>113</td>
</tr>
<tr>
<td>Error::Test()</td>
<td>114</td>
</tr>
<tr>
<td>Notes</td>
<td>114</td>
</tr>
<tr>
<td>Example</td>
<td>114</td>
</tr>
<tr>
<td>ErrorLog methods</td>
<td>115</td>
</tr>
<tr>
<td>ErrorLog::Abort()</td>
<td>115</td>
</tr>
<tr>
<td>Notes</td>
<td>115</td>
</tr>
<tr>
<td>Example</td>
<td>115</td>
</tr>
<tr>
<td>ErrorLog::Report()</td>
<td>116</td>
</tr>
<tr>
<td>Notes</td>
<td>116</td>
</tr>
<tr>
<td>Example</td>
<td>116</td>
</tr>
<tr>
<td>ErrorLog::SetLog(const char *)</td>
<td>117</td>
</tr>
<tr>
<td>Notes</td>
<td>117</td>
</tr>
<tr>
<td>Example</td>
<td>117</td>
</tr>
<tr>
<td>ErrorLog::SetSyslog()</td>
<td>118</td>
</tr>
<tr>
<td>Notes</td>
<td>118</td>
</tr>
<tr>
<td>Example</td>
<td>118</td>
</tr>
<tr>
<td>ErrorLog::SetTag(const char *)</td>
<td>119</td>
</tr>
<tr>
<td>Notes</td>
<td>119</td>
</tr>
<tr>
<td>Example</td>
<td>119</td>
</tr>
<tr>
<td>ErrorLog::UnsetSyslog()</td>
<td>120</td>
</tr>
<tr>
<td>Notes</td>
<td>120</td>
</tr>
<tr>
<td>Example</td>
<td>120</td>
</tr>
<tr>
<td>FileSys methods</td>
<td>121</td>
</tr>
<tr>
<td>FileSys::Chmod(FilePerm, Error *)</td>
<td>121</td>
</tr>
<tr>
<td>Notes</td>
<td>121</td>
</tr>
<tr>
<td>Example</td>
<td>121</td>
</tr>
<tr>
<td>FileSys::Close(Error *)</td>
<td>123</td>
</tr>
<tr>
<td>Notes</td>
<td>123</td>
</tr>
</tbody>
</table>
Signaler::OnIntr( SignalFunc, void * ) ................................................................. 169
  Notes ........................................................................................................... 169
  See also ....................................................................................................... 169
  Example ....................................................................................................... 170
Signaler::Signaler() (constructor) ................................................................. 171
  Notes ........................................................................................................... 171
  See also ....................................................................................................... 171

StrBuf methods ........................................................................................ 172
  StrBuf::Alloc( int ) ...................................................................................... 172
    Notes ....................................................................................................... 172
    Example .................................................................................................... 173
  StrBuf::Append( const char * ) .................................................................... 174
    Notes ....................................................................................................... 174
    Example .................................................................................................... 174
  StrBuf::Append( const char *, int ) ............................................................. 175
    Notes ....................................................................................................... 175
    Example .................................................................................................... 175
  StrBuf::Append( const StrPtr * ) ................................................................. 177
    Notes ....................................................................................................... 177
    Example .................................................................................................... 178
  StrBuf::Clear() .......................................................................................... 180
    Notes ....................................................................................................... 180
    See also .................................................................................................... 180
    Example .................................................................................................... 180
  StrBuf::StrBuf() (Constructor) .................................................................... 182
    Notes ....................................................................................................... 182
    Example .................................................................................................... 182
  StrBuf::StrBuf( const StrBuf & ) (Copy Constructor) ..................................... 183
    Notes ....................................................................................................... 183
    Example .................................................................................................... 183
  StrBuf::~StrBuf() (Destructor) ..................................................................... 184
    Notes ....................................................................................................... 184
    Example .................................................................................................... 184
  StrBuf::Extend( char ) .................................................................................. 185
    Notes ....................................................................................................... 185
    See also .................................................................................................... 185
    Example .................................................................................................... 185
  StrBuf::Extend( const char *, int ) ............................................................. 187
    Notes ....................................................................................................... 187
    See also .................................................................................................... 187
    Example .................................................................................................... 187
  StrBuf::operator =( const char * ) .............................................................. 189
    Notes ....................................................................................................... 189
    Example .................................................................................................... 189
  StrBuf::operator =( const StrBuf & ) .......................................................... 190
    Notes ....................................................................................................... 190
    Example .................................................................................................... 190
  StrBuf::operator ==( const StrPtr & ) .......................................................... 191
    Notes ....................................................................................................... 191
    Example .................................................................................................... 191
  StrBuf::operator ==( const StrRef & ) .......................................................... 192
    Notes ....................................................................................................... 192
    Example .................................................................................................... 192
  StrBuf::operator <=( const char * ) ............................................................. 193
<table>
<thead>
<tr>
<th>StrBuf:operator &lt;&lt;( int )</th>
<th>195</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes</td>
<td>195</td>
</tr>
<tr>
<td>Example</td>
<td>195</td>
</tr>
<tr>
<td>StrBuf:operator &lt;&lt;( const StrPtr * )</td>
<td>197</td>
</tr>
<tr>
<td>Notes</td>
<td>197</td>
</tr>
<tr>
<td>Example</td>
<td>197</td>
</tr>
<tr>
<td>StrBuf:operator &lt;&lt;( const StrPtr &amp; )</td>
<td>199</td>
</tr>
<tr>
<td>Notes</td>
<td>199</td>
</tr>
<tr>
<td>Example</td>
<td>200</td>
</tr>
<tr>
<td>StrBuf::Set( const char * )</td>
<td>201</td>
</tr>
<tr>
<td>Notes</td>
<td>201</td>
</tr>
<tr>
<td>Example</td>
<td>201</td>
</tr>
<tr>
<td>StrBuf::Set( const char *, int )</td>
<td>202</td>
</tr>
<tr>
<td>Notes</td>
<td>202</td>
</tr>
<tr>
<td>Example</td>
<td>202</td>
</tr>
<tr>
<td>StrBuf::Set( const StrPtr * )</td>
<td>203</td>
</tr>
<tr>
<td>Notes</td>
<td>203</td>
</tr>
<tr>
<td>Example</td>
<td>203</td>
</tr>
<tr>
<td>StrBuf::Set( const StrPtr &amp; )</td>
<td>204</td>
</tr>
<tr>
<td>Notes</td>
<td>204</td>
</tr>
<tr>
<td>Example</td>
<td>204</td>
</tr>
<tr>
<td>StrBuf::StringInit()</td>
<td>205</td>
</tr>
<tr>
<td>Notes</td>
<td>205</td>
</tr>
<tr>
<td>See also</td>
<td>205</td>
</tr>
<tr>
<td>Example</td>
<td>206</td>
</tr>
<tr>
<td>StrBuf::Terminate()</td>
<td>208</td>
</tr>
<tr>
<td>Notes</td>
<td>208</td>
</tr>
<tr>
<td>Example</td>
<td>208</td>
</tr>
<tr>
<td>See also</td>
<td>208</td>
</tr>
<tr>
<td>Example</td>
<td>209</td>
</tr>
<tr>
<td>StrDict methods</td>
<td>210</td>
</tr>
<tr>
<td>StrDict::GetVar( const StrPtr &amp; )</td>
<td>210</td>
</tr>
<tr>
<td>Notes</td>
<td>210</td>
</tr>
<tr>
<td>Example</td>
<td>210</td>
</tr>
<tr>
<td>StrDict::GetVar( const char * )</td>
<td>212</td>
</tr>
<tr>
<td>Notes</td>
<td>212</td>
</tr>
<tr>
<td>StrDict::GetVar( const char *, Error * )</td>
<td>213</td>
</tr>
<tr>
<td>Notes</td>
<td>213</td>
</tr>
<tr>
<td>StrDict::GetVar( const StrPtr &amp;, int )</td>
<td>214</td>
</tr>
<tr>
<td>Notes</td>
<td>214</td>
</tr>
<tr>
<td>StrDict::GetVar( const StrPtr &amp;, int, int )</td>
<td>215</td>
</tr>
<tr>
<td>Notes</td>
<td>215</td>
</tr>
<tr>
<td>StrDict::GetVar( int, StrPtr &amp;, StrPtr &amp; )</td>
<td>216</td>
</tr>
<tr>
<td>Notes</td>
<td>216</td>
</tr>
<tr>
<td>StrDict::Load( FILE * )</td>
<td>217</td>
</tr>
<tr>
<td>Notes</td>
<td>217</td>
</tr>
<tr>
<td>Example</td>
<td>217</td>
</tr>
<tr>
<td>StrDict::Save( FILE * )</td>
<td>218</td>
</tr>
<tr>
<td>Notes</td>
<td>218</td>
</tr>
<tr>
<td>Example</td>
<td>218</td>
</tr>
<tr>
<td>StrDict::SetArgv( int, char *const * )</td>
<td>219</td>
</tr>
<tr>
<td>Notes</td>
<td>219</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Notes</td>
<td>256</td>
</tr>
<tr>
<td>Example</td>
<td>256</td>
</tr>
<tr>
<td>StrRef::Set( char *, int )</td>
<td>257</td>
</tr>
<tr>
<td>Notes</td>
<td>257</td>
</tr>
<tr>
<td>Example</td>
<td>257</td>
</tr>
<tr>
<td>StrRef::Set( const StrPtr * )</td>
<td>258</td>
</tr>
<tr>
<td>Notes</td>
<td>258</td>
</tr>
<tr>
<td>Example</td>
<td>258</td>
</tr>
<tr>
<td>StrRef::Set( const StrPtr &amp; )</td>
<td>259</td>
</tr>
<tr>
<td>Notes</td>
<td>259</td>
</tr>
<tr>
<td>Example</td>
<td>259</td>
</tr>
</tbody>
</table>

**License Statements** ............................................................................................................. 261
About This Manual

This guide contains details about using the Perforce C/C++ API to create applications 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 C/C++, Java, Perl, Ruby, Python, PHP, and other languages are available from our website:

http://www.perforce.com/product/components/apis

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

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

Purpose of the API

The Perforce C/C++ API enables you to create applications that interact with end users, send commands to the Perforce server and process data returned from the versioning service. 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 is channeled through the ClientUser C++ class. The default methods of ClientUser implement the p4 command line interface. To create custom client applications, create subclasses based on ClientUser.

API files

The Perforce C/C++ 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:


and download p4api.tgz.

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

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

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

**Link order**

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

- `libclient.a`
- `librpc.a`
- `libsupp.a`
- `libp4sslstub.a`

In the Windows distribution, these files are named `libclient.lib`, `librpc.lib`, `libsupp.lib`, and `libp4sslstub.lib` respectively.

**SSL support**

The Perforce C/C++ API can be configured to support encrypted connections to the Perforce servers. To enable this support you must replace the bundled `libp4sslstub.a` (on Windows, `libp4sslstub.lib`) with copies of the OpenSSL libraries. (If you do not intend to use your application with a Perforce Server that supports encryption then you may simply compile the application with the supplied stub library.)

OpenSSL libraries are available from many sources; the most up-to-date is from [http://www.openssl.org/](http://www.openssl.org/).

**OpenSSL Library Version**

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

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

On Windows:

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

This command will produce an output similar to the following:

```
OpenSSL 1.0.1g 7 Apr 2014
```

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

**Link order for SSL support**

To enable SSL support, replace the stub with the ssl and crypto libraries from OpenSSL, resulting in the following link order:
• libclient.a
• librpc.a
• libsupp.a
• libssl.a
• libcrypto.a

On Windows, the ssl and crypto OpenSSL libraries are named \texttt{ssleay32.lib} and \texttt{libeay32.lib} respectively.

\textbf{Compiler support}

\textbf{UNIX}

For all UNIX platforms, you can use the \texttt{gcc} compiler to compile client applications with the Perforce C/C++ API. On Solaris, you can also use the Forte compiler.

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

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

\texttt{-lsocket -lnsl}

\textbf{Linux}

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

\texttt{-lrt}

\textbf{Windows}

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

\texttt{/DOS\_NT /MT /DCASE\_INSENSITIVE}

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

Link with the following libraries:

• libcmt.lib
• oldnames.lib
• kernel32.lib
• ws2_32.lib
• advapi32.lib

**Macintosh**

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

• Interfacelib
• PPCToolLibs.o
• PLStringFuncs PPC.lib
• MSL-MPWRuntime.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 `p4api.cc`, a Perforce application. (The example that the API is installed in the `api` subdirectory.)

```bash
C++FLAGS = -g -D_GNU_SOURCE ;
LINK = c++ ;OPTIM = ;
Main p4api : p4api.cc ;
ObjectHdrs p4api : api ;
LinkLibraries p4api : api/libclient.a api/librpc.a api/libsupp.a
          api/libp4sslstub.a;
```

For more about `jam`, see “Building with Jam” on page 6.

**Sample Makefile**

The following is a `gnumake` file for building `p4api.cc`, a Perforce application. (The example assumes the API is installed in the `api` subdirectory.)
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/documentation/jam

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

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

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

Compile and link p4api.cc as follows:

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

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

The preprocessor definitions (-Ddefinition) 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 how to build the sample application on each platform.

**Building the sample application**

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

```bash
$ 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)...
Cc++ p4api.o
Link p4api
Chmod1 p4api
...updated 2 target(s)...

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

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

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

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

Then use `jam -o build_sample` to create the build file:
$ jam -s OSVER=42 -o build_sample
...found 121 target(s)...
...updating 2 target(s)...
C++ p4api.o
Link p4api
Chmod1 p4api
...updated 2 target(s)...

$ cat build_sample

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

The generated build_sample can then be executed by a shell:

/bin/sh build_sample

to produce the executable, which you can test by running p4api info or most other Perforce commands:

$ p4api changes -m 1

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

As you can see, p4api is a usable full-featured command line 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 versioning service

Perforce applications interact with the versioning service 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 workspace. Different processes that use the same combination of user and workspace 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 user's 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 user's machine. Assuming the
workstation is configured correctly, your application does not need to provide logic that
specifies server, port, workspace, or user.

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

Settings take precedence as follows, highest to lowest:

1. Values set within a 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:

```c
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 10.

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
application, rather than during a single invocation. (For details about the `-o` and `-i` global
options, see the `P4 Command Reference`.)

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

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

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

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

### Sending commands

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

```cpp
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:

```cpp
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 application 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 add (2005.2)    p4 fixes (2000.1)    p4 protect -o
p4 branch -o      p4 group -o          p4 reviews (2005.2)
p4 branches       p4 groups (2004.2)      p4 reopen (2005.2)
p4 change -o (2005.2) p4 have (2005.2)    p4 resolve (2005.2)
p4 changes        p4 info (2003.2)        p4 revert (2005.2)
p4 client -o      p4 integrate (2005.2)  p4 review (2005.2)
p4 clients        p4 job -o             p4 submit (2005.2)
p4 counter (2005.2) p4 jobs              p4 sync (2005.2)
p4 counters (2000.2) p4 jobspec -o       p4 trigger -o
p4 delete (2005.2) p4 label -o          p4 typemap -o (2000.1)
p4 describe       p4 labels              p4 unlock (2005.2)
p4 depots (2005.2) p4 labelsync (2005.2) p4 user -o
p4 diff (2005.2)   p4 lock (2005.2)      p4 users
p4 filelog        p4 obliterate (2005.2)
p4 fix (2005.2)    p4 opened
```

The tagged output of some commands may have changed since the command's first appearance in this table. The output of `p4 resolve` and `p4 diff` are not fully tagged. For complete details, see the release notes:

http://www.perforce.com/perforce/r14.2/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 application has finished interacting with the Perforce server, it must disconnect as illustrated below:

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

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

Performing file I/O

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

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

```c
#include "p4/clientapi.h"
class MyFileSys : public FileSys {
    public:
        MyFileSys();
        ~MyFileSys();

        virtual void    Open( FileOpenMode mode, Error *e );
        virtual void    Write( const char *buf, int len, Error *e );
        virtual int     Read( char *buf, int len, Error *e );
        virtual int     ReadLine( StrBuf *buf, Error *e );
        virtual void    Close( Error *e );
        virtual int     Stat();
        virtual int     StatModTime();
        virtual void    Truncate( Error *e );
        virtual void    Unlink( Error *e = 0 );
        virtual void    Rename( FileSys *target, Error *e );
        virtual void    Chmod( FilePerm perms, Error *e );

    protected:
        int nchars;
    }

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

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

void MyFileSys::Open( FileOpenMode mode, Error *e )
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:

```c
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 Chapter 3, “Public Methods Reference” on page 21.

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.

**ClientProgress - progress indicators for Perforce commands**

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

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

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

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

**ClientUser - I/O for 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 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.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replacing the existing I/O routines is non-trivial. Your replacement routines must handle all special cases, including cross-platform file issues.</td>
</tr>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

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

**Ignore - support for rejecting files**

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

**KeepAlive - support for client-side disconnection**

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

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

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

Options - parse and store command line options

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

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

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 StrPtrs 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 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.
**ClientApi methods**

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

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>ClientApi</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>const char *c</code> the new <code>P4CLIENT</code> setting</td>
</tr>
<tr>
<td></td>
<td><code>Error *e</code> an <code>Error</code> object</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

**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.

```c
client.Init( &e );
client.DefineClient( "default_workspace", &e );
```
**ClientApi::DefineHost( const char *, Error * )**

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>ClientApi</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>const char *c</code> the new <code>P4HOST</code> setting</td>
</tr>
<tr>
<td></td>
<td><code>Error *e</code> an <code>Error</code> object</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

**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.

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


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

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

Virtual? No

Class ClientApi

Arguments

| const char *c | the new P4IGNORE setting |

| Error *e | an Error object |

Returns void

Notes

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

See also

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

Example

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

```c
#include "clientapi.h"

int main()
{
    ClientApi client;
    Error e;

    client.Init( &e );
    client.DefineIgnoreFile( ".p4ignore", &e );
}
```
ClientApi::DefinePassword( const char *, Error * )

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

Virtual? No

Class ClientApi

Arguments

| const char *c | the new P4PASSWD setting |
| Error *e | an Error object |

Returns void

Notes

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

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

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

Example

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

```c
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

<table>
<thead>
<tr>
<th>Class</th>
<th>ClientApi</th>
</tr>
</thead>
</table>

**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.)

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *c  the new P4USER setting</td>
</tr>
<tr>
<td></td>
<td>Error *e  an Error object</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

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.

```c
client.Init( &e );
client.DefineUser( "default_user", &e );
```
ClientApi::Dropped()

Check if connection is no longer usable.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>int nonzero if the connection has dropped</td>
</tr>
</tbody>
</table>

Notes

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

Example

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

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

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>Error *e</td>
</tr>
<tr>
<td>Returns</td>
<td>int</td>
</tr>
</tbody>
</table>

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 p4api.cc, and reports the number of errors before the program exits:

```
client.Init( &e );
client.SetArgv( argc - 2, argv + 2 );
client.Run( argv[1], &ui );
printf( "There were %d errors.\n", client.Final( &e ) );
```
### ClientApi::GetClient()

Get current client setting.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>const StrPtr &amp; a reference to the client setting</td>
</tr>
</tbody>
</table>

**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 `StrPtrs` and `StrBufs`.

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

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

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

Executing the preceding code produces the following output:

```
Current client two = two
Previous client setting was one
```
**ClientApi::GetConfig()**

Get current configuration file.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>const StrPtr &amp; a reference to the config file setting</td>
</tr>
</tbody>
</table>

**Notes**

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

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

**Example**

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

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

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

```
C:\perforce> a.out
Current P4CONFIG is noconfig
```
**ClientApi::GetCwd()**

Get current working directory.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>const StrPtr &amp; a reference to the name of the current directory</td>
</tr>
</tbody>
</table>

**Notes**

See [GetClient()](#) for more about the StrPtr return value.

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

**Example**

The following example demonstrates the usage of [GetCwd()](#).

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>const StrPtr &amp;  a reference to the hostname</td>
</tr>
</tbody>
</table>

**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()`.

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

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

Executing the preceding code produces the following output:

```
shire% a.out
Client hostname is shire
```
ClientApi::GetIgnore()

Virtual? No

Class ClientApi

Arguments None

Returns Ignore *i an Ignore object, which can be used to determine if a path is ignored.

Notes

If P4IGNORE is not set, no paths are ignored.

See also

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

Example

This example demonstrates the use of GetIgnore().

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

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>const StrPtr &amp;</td>
</tr>
</tbody>
</table>

**Notes**

See [GetClient()](#) for more about the StrPtr return value.

If the P4IGNORE is unset, [GetIgnoreFile()](#) returns an uninitialized StrPtr.

**See also**

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

**Example**

This example demonstrates the use of [GetIgnoreFile()](#).

```c
#include "clientapi.h"

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

Executing the preceding code produces output similar to the following:

The current ignore file is .p4ignore
ClientApi::GetOs()

Get name of client operating system.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>const StrPtr &amp; a reference to the OS string</td>
</tr>
</tbody>
</table>

**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()**.

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>const StrPtr &amp; a reference to the password</td>
</tr>
</tbody>
</table>

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().

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td>ClientApi</td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>const StrPtr &amp;  a reference to the port setting</td>
</tr>
</tbody>
</table>

**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()](#).

```cpp
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>ClientApi</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *v the name of the protocol variable being checked</td>
</tr>
<tr>
<td>Returns</td>
<td>StrPtr * a pointer to the variable's value</td>
</tr>
</tbody>
</table>

**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.

```c
...  
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().

```cpp
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>Error *e an Error object</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

Notes

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

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

Example

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

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

client.Init( &e );
client.SetArgv( argc - 2, argv + 2 );
client.Run( argv[1], &ui );
client.Final( &e );
return 0;
```
ClientApi::Run( const char *, ClientUser * )

Run a Perforce command and return when it completes.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>ClientApi</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
<th>const char *func</th>
<th>the name of the command to run</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ClientUser *ui</td>
<td>a pointer to a ClientUser object.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>void</th>
</tr>
</thead>
</table>

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.

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.

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

client.Init( &e );
client.Run( "info", &ui );
client.Final( &e );
```
ClientApi::SetBreak( KeepAlive *breakCallback )

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>KeepAlive *breakCallback  keepalive callback for user interrupt</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

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.
```cpp
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr *c the new client setting</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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.

```c
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.

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td><strong>ClientApi</strong></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>*<em>const StrPtr <em>c</em></em> the new directory path</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td><strong>void</strong></td>
</tr>
</tbody>
</table>

**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`.

```c++
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>ClientApi</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *c the new directory path</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**.

```c
ClientApi client;
ClientUser ui;

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

client.SetCwd( "C:\two" );
client.Run( "info", &ui );
```
**ClientApi::SetCwdNoReload( const StrPtr * )**

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr *c the new directory path</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

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

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

**Example**

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

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

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

client.SetCwdNoReload( &sb2 );
client.Run( "info", &ui );
```
**ClientApi::SetCwdNoReload( const char * )**

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td><strong>ClientApi</strong></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td><code>const char *c</code> the new directory path</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

**Notes**

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

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

**Example**

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

```c
ClientApi client;
ClientUser ui;

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

client.SetCwdNoReload( "C:\two" );
client.Run( "info", &ui );
```
### ClientApi::SetHost( const StrPtr * )

Sets the hostname to be used for this connection.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr *c the new hostname value</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

#### 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`.

```cpp
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *c the new hostname value</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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`.

```c
ClientApi client;
ClientUser ui;

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

client.SetHost( "shire" );
client.Run( "info", &ui );
```
ClientApi::SetIgnoreFile( const StrPtr * )

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

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

Notes

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

See also

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

Example

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

```c
#include "clientapi.h"

int main()
{
    ClientApi client;
    StrBuf sb;
    sb = ".p4ignore";
    client.SetIgnoreFile( &sb; );
}
```
### ClientApi::SetIgnoreFile( const char * )

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *c the full path name of the new ignore file</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

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

**See also**

- ClientApi::DefineIgnoreFile()
- ClientApi::GetIgnore()
- ClientApi::GetIgnoreFile()

**Example**

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

```c
#include "clientapi.h"

int main()
{
  ClientApi client;
  client.SetIgnoreFile( ".p4ignore" );
}
```
ClientApi::SetPassword( const StrPtr * )

Sets the password to be used for this connection.

Virtual? No

Class  ClientApi

Arguments  const StrPtr *c  the new password value

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

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *c the new password value</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

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

**Example**

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

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr *c the new port value</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

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.

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

sb = "ssl:magic:1666";
client.SetPort( &sb );
client.Init( &e );
```
**ClientApi::SetPort( const char * )**

Sets the port to be used to open this connection.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>ClientApi</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
<th>const char *c the new port value</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>void</th>
</tr>
</thead>
</table>

**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.

```cpp
ClientApi client;
Error e;

client.SetPort( "magic:1666" );
client.Init( &e );
```
**ClientApi::SetProg( const StrPtr * )**

Sets the application or script name for this connection.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>ClientApi</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr *c the new program name</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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()`.

**See also**

`ClientApi::SetVersion()`

**Example**

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

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td>ClientApi</td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>const char *c</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>void</td>
</tr>
</tbody>
</table>

**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().

**See also**

ClientApi::SetVersion()

**Example**

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

```c
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():

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>To enable tagged output (if tagged output for the command is supported by the server), set the tag variable to any value.</td>
</tr>
<tr>
<td>specstring</td>
<td>To enable specially formatted application forms, set the specstring to any value.</td>
</tr>
<tr>
<td>api</td>
<td>Set the api variable to the value corresponding to the level of server behavior your application supports.</td>
</tr>
</tbody>
</table>

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

For instance, the "p4 info" command supports tagged output as of server release 2003.2, and changes to this format were made in 2004.2. Code requesting tagged output from "p4 info" that was compiled against the 2003.1 API library 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Set api to</th>
<th>Tagged output supported?</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>unset</td>
<td>Only if both server and API are at 2004.2 or greater</td>
</tr>
<tr>
<td></td>
<td>&lt;=55</td>
<td>Output is not tagged; behaves like 2003.1 or earlier, even if server supports tagged output.</td>
</tr>
<tr>
<td></td>
<td>=56</td>
<td>Output is tagged; behaves like 2003.2.</td>
</tr>
</tbody>
</table>
Command | Set api to | Tagged output supported?
---|---|---
| =58 | Output is tagged; behaves like 2005.2 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()`.

```c
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.

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>char *nv</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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()*.

```c
ClientApi client;
Error e;

client.SetProtocolV( "tag=" );
client.Init( &e );
client.Run( "branches", &ui );
client.Final( &e );
```
ClientApi::SetTicketFile( const StrPtr * )

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr *c the full path name of the new ticket file</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

Notes

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

Example

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

```c++
ClientApi client;
StrBuf sb;

sb = "/tmp/ticketfile.txt";
client.SetTicketFile( &sb );
```
**ClientApi::SetTicketFile( const char * )**

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *c the full path name of the new ticket file</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

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

**Example**

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

```cpp
ClientApi client;
client.SetTicketFile( "/tmp/ticketfile.txt" );
```
**ClientApi::SetUi( ClientUser *)**

Reset the ClientUser object used for this connection.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>ClientUser *ui  a pointer to a ClientUser object.</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

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

**Example**

The following example illustrates two ways to run p4 info:

```cpp
ClientApi client;
ClientUser ui;

client.Run( "info", &ui );
client.SetUi( &ui );
client.Run( "info" );
```
**ClientApi::SetUser( const StrPtr * )**

Sets the user for this connection.

**Virtual?**  No

**Class**  ClientApi

**Arguments**  const StrPtr *c  the new user name setting

**Returns**  void

**Notes**

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

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

**Example**

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

```cpp
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.

```c
ClientApi client;
Error e;

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

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

client.SetUser( "user2" );
client.SetArgv( 1, args );
client.Run( "user", &ui );
```
**ClientApi::SetVersion( const StrPtr * )**

Sets the application or script version for this connection.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr *c the new version number</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

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

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

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

**See also**

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

**Example**

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

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

sb.Set( "2005.2" );

client.Init( &e );
client.SetVersion( &sb );
client.Run( "info", &ui );
```
**ClientApi::SetVersion( const char * )**

Sets the application or script version for this connection.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *c</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

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

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

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

**See also**

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

**Example**

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

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

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

ClientProgress::Description( const StrPtr *, int )

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

Virtual? Yes
Class ClientProgress

Arguments

- const StrPtr *desc
  description from the server
- int units
  the units in which progress is to be measured

Returns void

Notes

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

<table>
<thead>
<tr>
<th>Client Progress Unit</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_UNSPECIFIED</td>
<td>0</td>
<td>No units specified</td>
</tr>
<tr>
<td>CPU_PERCENT</td>
<td>1</td>
<td>Value is a percentage</td>
</tr>
<tr>
<td>CPU_FILES</td>
<td>2</td>
<td>Value is a count of files</td>
</tr>
<tr>
<td>CPU_KBYTES</td>
<td>3</td>
<td>Value is in kilobytes</td>
</tr>
<tr>
<td>CPU_MBYTES</td>
<td>4</td>
<td>Value is in megabytes</td>
</tr>
</tbody>
</table>

See also

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

Example

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

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

Called when an operation completes.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientProgress</td>
</tr>
<tr>
<td>Arguments</td>
<td>int fail</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

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

**See also**

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

**Example**

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

```c++
void MyProgress::Done( int fail )
{
    printf( fail ? "Command failed\n" : "Command completed\n");
}
```
ClientProgress::Total( long )

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientProgress</td>
</tr>
<tr>
<td>Arguments</td>
<td>long units Total number of client progress units expected, if known</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

Notes

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

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

See also

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

Example

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

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

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

Reports on command progress and user cancellation requests.

Virtual?   Yes

Class        ClientProgress

Arguments  long units   Total number of progress units processed, if known

Returns   int

Notes

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

In addition to reporting progress in terms of the units defined by Description(), if Update() returns non-zero, the API interprets it as a user request to cancel the operation.

See also

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

Example

To report on units processed, create a subclass of ClientProgress and define an alternate implementation of Update(). A trivial implementation ignores cancel requests by always returning 0; a more useful implementation might resemble the following:

```c
void MyProgress::Update( long units )
{
    if ( cancelclicked() ) // has anyone clicked the Cancel button?
    {
        return 1; // yes, user wishes to cancel
    }
    else
    {
        displayGUI( units ); // show how many units have been processed
        return 0; // user has not requested cancel, continue processing
    }
}
```
**ClientUser methods**

**ClientUser::CreateProgress(int)**

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

**Virtual?** Yes

**Class** `ClientUser`

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProgressType</td>
<td><code>int</code></td>
<td>the type of progress to be reported</td>
</tr>
</tbody>
</table>

**Returns**

*`ClientProgress`* a pointer to the new `ClientProgress` object.

**Notes**

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

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

<table>
<thead>
<tr>
<th>Client Progress Type</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPT_SENDFILE</td>
<td>1</td>
<td>Files sent to server</td>
</tr>
<tr>
<td>CPT_RECVFILE</td>
<td>2</td>
<td>Files received from server</td>
</tr>
<tr>
<td>CPT_FILESTRANS</td>
<td>3</td>
<td>Files transmitted</td>
</tr>
<tr>
<td>CPT_COMPUTATION</td>
<td>4</td>
<td>Computation performed server-side</td>
</tr>
</tbody>
</table>

**See also**

- `ClientUser::ProgressIndicator()`  
- `ClientProgress::Description()`  
- `ClientProgress::Done()`  
- `ClientProgress::Total()`  
- `ClientProgress::Update()`
**ClientUser::Diff( FileSys *, FileSys *, int, char *, Error * )**

Diff two files, and display the results.

### Virtual?
Yes

### Class
**ClientUser**

### Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileSys *f1</td>
<td>the first file to be diffed</td>
</tr>
<tr>
<td>FileSys *f2</td>
<td>the second file to be diffed</td>
</tr>
<tr>
<td>int doPage</td>
<td>should output be paged?</td>
</tr>
<tr>
<td>char *diffFlags</td>
<td>flags to diff routine</td>
</tr>
<tr>
<td>Error *e</td>
<td>an Error object</td>
</tr>
</tbody>
</table>

### 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 C/C++ API); otherwise, `Diff()` simply calls the specified external program.

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

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

### See also

`ClientUser::RunCmd()`

### Example

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

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

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>a temp file containing the head revision of depot file <code>file.c</code></td>
</tr>
<tr>
<td>f2</td>
<td>the local workspace version of file <code>file.c</code></td>
</tr>
<tr>
<td>doPage</td>
<td>0</td>
</tr>
<tr>
<td>diffFlags</td>
<td>c</td>
</tr>
</tbody>
</table>
The diff is performed by creating a `Diff` object, giving it `f1` and `f2` as its inputs, and `-c` as its flag. The end result is sent to `stdout`. If either of the files is binary, the message "files differ" is printed instead.

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientUser</td>
</tr>
<tr>
<td>Arguments</td>
<td>FileSys *f1</td>
</tr>
<tr>
<td></td>
<td>FileSys *f2</td>
</tr>
<tr>
<td></td>
<td>FileSys *fout</td>
</tr>
<tr>
<td></td>
<td>int doPage</td>
</tr>
<tr>
<td></td>
<td>char *diffFlags</td>
</tr>
<tr>
<td></td>
<td>Error *e</td>
</tr>
</tbody>
</table>

Returns void

Notes

This method works like Diff(), but instead of sending data to the standard output, writes the data to the specified output file.
**ClientUser::Edit( FileSys *, Error *)**

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientUser</td>
</tr>
<tr>
<td>Arguments</td>
<td>FileSys *f1 the file to be edited</td>
</tr>
<tr>
<td></td>
<td>Error *e an Error object</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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:

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

Outputs an error and prompts for a keystroke to continue.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientUser</td>
</tr>
<tr>
<td>Arguments</td>
<td>char *errBuf</td>
</tr>
<tr>
<td></td>
<td>Error *e</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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:

```c
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

<table>
<thead>
<tr>
<th>Class</th>
<th>ClientUser</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
<th>FileSysType type</th>
<th>the file type of the file to be created</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>FileSys *</th>
<th>a pointer to the new FileSys.</th>
</tr>
</thead>
</table>

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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientUser</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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 application to beep after each command, put the command into `Finished()`, as follows.

```c
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:

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

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>ClientUser</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>const char *const *help</code> an array of arrays containing the help text.</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

**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:

```c
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`.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientUser</td>
</tr>
<tr>
<td>Arguments</td>
<td>StrBuf *strbuf the StrBuf which is to hold the data</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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`.

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientUser</td>
</tr>
<tr>
<td>Arguments</td>
<td>FileSys *base</td>
</tr>
<tr>
<td></td>
<td>FileSys *leg1</td>
</tr>
<tr>
<td></td>
<td>FileSys *leg2</td>
</tr>
<tr>
<td></td>
<td>FileSys *result</td>
</tr>
<tr>
<td></td>
<td>Error *e</td>
</tr>
</tbody>
</table>

**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 P4Merge as well as third-party tools). External merge programs must be specified by an environment variable, either `P4MERGE` or `MERGE`. `Merge()` returns after the external merge program exits.

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

**See also**

`ClientUser::RunCmd()`

**Example**

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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>A temp file built from the depot revision that is the &quot;base&quot; of the resolve.</td>
</tr>
<tr>
<td>leg1</td>
<td>A temp file built from the depot revision that is the &quot;theirs&quot; of the resolve.</td>
</tr>
<tr>
<td>leg2</td>
<td>The local workspace file that is the &quot;yours&quot; of the resolve.</td>
</tr>
<tr>
<td>result</td>
<td>A temp file in which to construct the new revision of &quot;yours&quot;.</td>
</tr>
</tbody>
</table>

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

After you "accept" the merged file (with "ae"), the "result" temp file is copied into the "leg2" or "yours" workspace file, and this is the file that is submitted to the depot.
To change the way that external merge programs are called during a resolve, create a subclass of `ClientUser` with an alternate implementation of `Merge()`.

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

```c++
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>ClientUser</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td>Error *e</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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 application to be compatible with pre-2002.1 servers, use this default implementation of `Message()` - newer servers will call `Message()`, and older servers will call `OutputInfo()` and `HandleError()` directly.

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

**Example**

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

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

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

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

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

```c
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:

```c
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:

```cpp
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientUser</td>
</tr>
<tr>
<td>Arguments</td>
<td>StrDict *varList</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

#### 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
... isMapped
... headAction integrate
... headType text
... headTime 998644337
... headRev 10
... headChange 681
... headModTime 998643970
... haveRev 10
```

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

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

```c++
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

```plaintext
> 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.

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

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClientUser</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>int returns non-zero if progress indicators are desired, 0 otherwise</td>
</tr>
</tbody>
</table>

**Notes**

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

**See also**

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

**Example**

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

```cpp
MyUserProgress::ProgressIndicator()
{
  return 1;
}
```
ClientUser::Prompt( const StrPtr &, StrBuf &, int, Error * )

Prompt the user and get a response.

Virtual? Yes

Class ClientUser

Arguments

const StrPtr &msg the message with which to prompt the user

StrBuf &rsp where to put the user’s response

int noEcho specifies whether echo should be turned off at the console

Error *e an Error object

Returns void

Notes

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

Example

Consider the following user interaction with p4 resolve:

```
> p4 resolve file.c
C:\depot\file.c - merging //depot/file.c#2,#10
Diff chunks: 0 yours + 1 theirs + 0 both + 0 conflicting
Accept(a) Edit(e) Diff(d) Merge (m) Skip(s) Help(?) [at]: at
```

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

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

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

```cpp
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>ClientUser</strong> (static)</td>
</tr>
</tbody>
</table>

#### 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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>P4DIFF</td>
</tr>
<tr>
<td>arg1</td>
<td>local file name</td>
</tr>
<tr>
<td>arg2</td>
<td>temp file name (depot file)</td>
</tr>
<tr>
<td>arg3</td>
<td>null</td>
</tr>
<tr>
<td>arg4</td>
<td>null</td>
</tr>
<tr>
<td>pager</td>
<td>P4PAGER</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>Error</code></td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

### 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:

```c++
ClientApi client;
Error e;

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

printf( "Error severity after Init() 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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Error</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char * trace a string to appear next to the debugging output</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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:

```c
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`.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>Error</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>StrBuf *buf</code> a pointer to the <code>StrBuf</code> to contain the formatted error</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

**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:

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><em>Error</em></td>
</tr>
</tbody>
</table>

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>StrBuf *buf</td>
<td></td>
<td>a pointer to the <em>StrBuf</em> to contain the formatted error</td>
</tr>
<tr>
<td>int opts</td>
<td></td>
<td>formatting options</td>
</tr>
</tbody>
</table>

**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:

- **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.

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Error</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>int</td>
</tr>
</tbody>
</table>

**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.

```cpp
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Error</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>int</td>
</tr>
</tbody>
</table>

Notes

The severity can take the following values:

<table>
<thead>
<tr>
<th>Severity</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_EMPTY (0)</td>
<td>no error</td>
</tr>
<tr>
<td>E_INFO (1)</td>
<td>information, not necessarily an error</td>
</tr>
<tr>
<td>E_WARN (2)</td>
<td>a minor error occurred</td>
</tr>
<tr>
<td>E_FAILED (3)</td>
<td>the command was used incorrectly</td>
</tr>
<tr>
<td>E_FATAL (4)</td>
<td>fatal error, the command can't be processed</td>
</tr>
</tbody>
</table>

Example

The following code attempts to establish a connection to a server, and beeps if the severity is a warning or worse:

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Error</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>int nonzero if error is fatal</td>
</tr>
</tbody>
</table>

**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:

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Error</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>int nonzero if the most severe error is a warning</td>
</tr>
</tbody>
</table>

**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:

```c
ClientApi client;
Error e;

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

if ( e.IsWarning() ) printf( "Warning!\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.

```c
    e.Net( "bind", service.Text() );
```
Error::operator << ( int )

Add data to the text of an error message.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Error</td>
</tr>
<tr>
<td>Arguments</td>
<td>int arg</td>
</tr>
<tr>
<td>Returns</td>
<td>Error &amp;</td>
</tr>
</tbody>
</table>

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.

```c++
e.Set( E_WARN, "Warning, number " ) << myErrNum;
```
**Error::operator << ( char * )**

Add data to the text of an error message.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Error</td>
</tr>
<tr>
<td>Arguments</td>
<td>char *arg text to be added to this Error</td>
</tr>
<tr>
<td>Returns</td>
<td>Error &amp; a reference to the modified Error</td>
</tr>
</tbody>
</table>

**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.

```plaintext
  e.Set( E_WARN, "Warning! " ) << "Something bad happened"
```
**Error::operator << ( const StrPtr &)**

Add data to the text of an error message.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Error</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr &amp;arg  text to be added to this Error</td>
</tr>
<tr>
<td>Returns</td>
<td>Error &amp; a reference to the modified Error</td>
</tr>
</tbody>
</table>

**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.

```c++
Error e1, e2;
e1 = e2;
```
**Error::Set( enum ErrorSeverity, const char * )**

Add an error message to an Error.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Error</td>
</tr>
<tr>
<td>Arguments</td>
<td>enum ErrorSeverity s</td>
</tr>
<tr>
<td></td>
<td>const char *fmt</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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.

```c
Error e;
e.Set( E_FATAL, "Fatal error!");
```
**Error::Set( ErrorId & )**

Add an error message to an Error.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Error</td>
</tr>
<tr>
<td>Arguments</td>
<td>ErrorId&amp; id</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td><strong>Error</strong></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>*<em>const char <em>op</em></em></td>
</tr>
<tr>
<td></td>
<td>*<em>const char <em>arg</em></em></td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>void</td>
</tr>
</tbody>
</table>

**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.

```c
    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:

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ErrorLog</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

Notes

If the error is empty (severity is E_EMPTY), Abort() returns. Otherwise Abort() causes the program to exit with a status of -1.

Example

Abort() is typically called after Init() or Run() to abort the program with a non-zero status if there has been a connection problem. The code in p4api.cc is one example:

```cpp
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`.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ErrorLog</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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.

```cpp
ClientApi client;
Error e;

client.Init( &e );
ErrorLog::Report();
```
**ErrorLog::SetLog( const char * )**

Redirects this Error's **Report()** output to a file.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>ErrorLog</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>const char *file</code>  the file to serve as an error log</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

**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.

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ErrorLog</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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.

```cpp
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ErrorLog</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *tag the text of the new tag</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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".

```c++
ClientApi client;
Error e;

client.Init( &e );
ErrorLog::SetTag( "NewError" );
```
**ErrorLog::UnsetSyslog()**

Stop writing errors to *syslog*.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>ErrorLog</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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.

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
</table>

| Class         | FileSys              |

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilePerm perms</td>
<td>permissions to change the file, either FPM_RO (read only) or FPM_RW (read/write)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error *error</th>
<th>returned error status</th>
</tr>
</thead>
</table>

| 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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPM_RO</td>
<td>0x00</td>
<td>leave file read-only.</td>
</tr>
<tr>
<td>FPM_RW</td>
<td>0x01</td>
<td>allow read and write operations</td>
</tr>
</tbody>
</table>

**Example**

To use `Chmod()` to create a configuration file and set its permissions to read-only:

```c
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:

```cpp
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:

```cpp
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>FileSys</td>
</tr>
<tr>
<td>Arguments</td>
<td>FileSysType type file type</td>
</tr>
<tr>
<td>Returns</td>
<td>FileSys * a pointer to the new FileSys.</td>
</tr>
</tbody>
</table>

**Notes**

A FileSysType is an enum taking one of the values defined in `filesys.h`. The most commonly used FileSysTypes are as follows:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>FST_TEXT</td>
<td>0x0001</td>
<td>file is text</td>
</tr>
<tr>
<td>FST_BINARY</td>
<td>0x0002</td>
<td>file is binary</td>
</tr>
<tr>
<td>FST_ATEXT</td>
<td>0x0011</td>
<td>file is text, open only for append</td>
</tr>
</tbody>
</table>

**Example**

To use `Create()` to create a FileSys object for a log file (text file, append-only):

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>FileSys</td>
</tr>
<tr>
<td>Arguments</td>
<td>FileOpenMode mode Mode to open the file, either FOM_READ (open for read) or FOM_WRITE (open for write)</td>
</tr>
<tr>
<td></td>
<td>Error *error returned error status</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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:

```cpp
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'
            Name(),
            mode == FOM_READ ? "read" : "write" );
    }
}
**FileSys::Read( const char *, int, Error * )**

Attempt to read \texttt{len} bytes of data from the object referenced by the file handle (returned by the \texttt{Open()} method) to the buffer pointed to by \texttt{buf}. Upon successful completion, \texttt{Read()} returns the number of bytes actually read and placed in the buffer.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td>\texttt{FileSys}</td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>\texttt{const char *buf}</td>
</tr>
<tr>
<td></td>
<td>\texttt{int len}</td>
</tr>
<tr>
<td></td>
<td>\texttt{Error *error}</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>\texttt{int}</td>
</tr>
</tbody>
</table>

**Notes**

The default implementation of \texttt{Read()} is called every time there is a need to read data from the file referenced by the \texttt{Open()} call.

Your implementation must correctly report any system errors that may occur during I/O.

**Example**

To use \texttt{Read()} to read a line from a log file:

```c
char line[80];
msg.Set( );
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 \texttt{Read()} to report errors with \texttt{Error::Sys()}, provide debugging output, and use the \texttt{FileSysDemo} member "fd" to hold the file handle returned from the \texttt{read()} system call:
```c
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:

```cpp
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:

```cpp
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>FileSys</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>const StrPtr *name</code> filename for this <code>FileSys</code> object</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

**Notes**

After creating a `FileSys` object, call `Set()` to supply it with a `path`.

**Example**

To use `Set()` to set a filename:

```c
FileSys *f = FileSys::Create( FST_BINARY );
f->Set( "/tmp/file.bin" );
```

To reimplement `Set()` to provide debugging output:

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>FileSys</code></td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td><code>int</code> 0 for failure, or status bits as defined below</td>
</tr>
</tbody>
</table>

The status bits have the following meanings:

<table>
<thead>
<tr>
<th>Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>failure</td>
</tr>
<tr>
<td>FSF_EXISTS (0x01)</td>
<td>file exists</td>
</tr>
<tr>
<td>FSF_WRITEABLE (0x02)</td>
<td>file is user-writable</td>
</tr>
<tr>
<td>FSF_DIRECTORY (0x04)</td>
<td>file is a directory</td>
</tr>
<tr>
<td>FSF_SYMLINK (0x08)</td>
<td>file is symlink</td>
</tr>
<tr>
<td>FSF_SPECIAL (0x10)</td>
<td>file is a special file (in the UNIX sense)</td>
</tr>
<tr>
<td>FSF_EXECUTABLE (0x20)</td>
<td>file is executable</td>
</tr>
<tr>
<td>FSF_EMPTY (0x40)</td>
<td>file is empty</td>
</tr>
<tr>
<td>FSF_HIDDEN (0x80)</td>
<td>file is invisible (hidden)</td>
</tr>
</tbody>
</table>

**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`:

```c++
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>FileSys</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>int 0 for failure, or last modified time in seconds since 00:00:00, January 1, 1970, GMT.</td>
</tr>
</tbody>
</table>

**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:

```c
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:

```c
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 );
}
```
**FileSync::Truncate()**

Truncate the file specified by the path protected FileSys member to zero length.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>FileSys</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>FileSys</td>
</tr>
<tr>
<td>Arguments</td>
<td>Error *error returned error status</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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:

```cpp
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:

```cpp
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".

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>FileSys</td>
</tr>
</tbody>
</table>

**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:

```c
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:

```c
void FileSysDemo::Write( const char *buf, int len, Error *e )
{
    int bytes;
    
    if ( ( bytes = write( fd, buf, len ) ) < 0 )
        e->Sys( "write", Name() );
    
    if ( DEBUG )
        printf( "debug (Write): %d bytes\n", bytes );
}
```
## Ignore methods

### Ignore::Reject( const StrPtr &path, const StrPtr &ignoreFile )

Tests whether the provided path will be rejected when it is opened for add because it matches an entry in the provided ignore file.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Ignore</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr &amp;path, const StrPtr &amp;ignoreFile</td>
</tr>
<tr>
<td>Returns</td>
<td>int</td>
</tr>
</tbody>
</table>

### Notes

Calling `Reject()` provides a preview of what will happen when files are opened for add.

If the ignore file does not exist, or is not readable, no files are rejected.

### Example

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

```c
#include "clientapi.h"
#include "ignore.h"

int main()
{
    ClientApi client;
    StrBuf clientPath;

    client.SetIgnoreFile( ".p4ignore" );
    clientPath = "ignore.txt";
    if ( client->GetIgnore()->Reject( *clientPath, client->GetIgnoreFile() ) )
    {
        printf( "%s is to be ignored.\n", clientPath.Text() );
    }
}
```
Ignore::RejectCheck( const StrPtr & )

Tests whether the provided path will be rejected when it is opened for add because it matches an ignore file entry.

Virtual? No

Class Ignore

Arguments cont StrPtr &path the path to check

Returns int nonzero if path is ignored

Notes

Calling RejectCheck() provides a preview of what will happen the file is opened for add.

Use RejectCheck() when you have to test multiple paths that may be rejected. First call Reject() to parse the ignore file, and then call RejectCheck() for each additional path that needs to be checked.

Example

The following example demonstrates the usage of RejectCheck().

```c
#include "clientapi.h"
#include "ignore.h"

int main()
{
    ClientApi client;
    StrBuf clientPath;

    client.SetIgnoreFile( ".p4ignore" );
    clientPath = "ignore.txt";
    if ( client->GetIgnore()->Reject( *clientPath,
            client->GetIgnoreFile() ) )
    {
        printf( "%s is to be ignored.\n", clientPath.Text() );
    }

    clientPath = "ignore2.txt";
    if ( client->GetIgnore()->Reject( *clientPath,
            client->GetIgnoreFile() ) )
    {
        printf( "%s is to be ignored.\n", clientPath.Text() );
    }
}
```
KeepAlive methods

**KeepAlive::IsAlive()**

The only method of the `KeepAlive` class, `IsAlive()` is used in applications to request that the current command be terminated by disconnecting.

| Virtual? | Yes |
| Class    | `KeepAlive` |
| Arguments| None |
| Returns  | `int` | 0 to terminate connection, 1 to continue processing |

**Notes**

Use `ClientApi::SetBreak()` to establish a callback to be called every 0.5 seconds during command execution.

**See also**

`ClientApi::SetBreak()`

**Example**

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

#include <clientapi.h>

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

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

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

client.Init( &e );
client.SetBreak( &cb );  // SetBreak must happen after the Init
client.Run( "changes", &ui );
client.Final( &e );
MapApi methods

MapApi::Clear()

Empties a mapping.

Virtual?  No
Class  MapApi
Arguments  None
Returns  void

Notes

After this method has been called on a MapApi object, the object is indistinguishable from a freshly-constructed object.
**MapApi::Count()**

Returns the number of entries currently in the mapping.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual?</td>
<td>No</td>
</tr>
<tr>
<td>Class</td>
<td>MapApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>int</td>
</tr>
</tbody>
</table>

**Notes**

The number returned by `Count()` may be different from the number of times that `Insert()` has been called. This is because `MapApi` automatically disambiguates itself, adding new exclusions to eliminate ambiguity between partially overlapping entries and removing entries that are redundant.

**Example**

The following example demonstrates `Count()`, `GetType()`, `GetLeft()`, and `GetRight()` being used to iterate over a `MapApi` that contains four entries after two calls to `Insert()`.

This code produces the following output:

```c
//depot/... //client/...
-//depot/d2/... //client/d2/...
-//depot/d1/... //client/d1/...
//depot/d1/... //client/d2/...
```

```c
MapApi clientmap;
clientmap.Insert( StrRef( "//depot/..." ), StrRef( "//client/..." ) );
clientmap.Insert( StrRef( "//depot/d1/..." ), StrRef( "//client/d2/..." ) );

char c = ' ';
for ( int i = 0; i < clientmap.Count(); i++ )
{
    switch( clientmap.GetType( i ) )
    {
        case MapInclude:
            c = ' '; break;
        case MapExclude:
            c = '-'; break;
        case MapOverlay:
            c = '+'; break;
    }
    printf( "%c%s %s\n", c,
            clientmap.GetLeft( i )->Text(),
            clientmap.GetRight( i )->Text() );
}
```
MapApi::GetLeft( int )

Returns the left side of the specified view entry.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>MapApi</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
<th>int i</th>
<th>the index of the desired entry</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>const StrPtr *</th>
<th>a string representing the left side of the entry</th>
</tr>
</thead>
</table>

Notes

The index should be between 0 and one less than the number of mapping entries.

See also

MapApi::Count()

Example

See the example for MapApi::Count().
MapApi::GetRight( int )

Returns the right side of the specified view entry.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>MapApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>int i</td>
</tr>
<tr>
<td></td>
<td>the index of the desired entry</td>
</tr>
<tr>
<td>Returns</td>
<td>const StrPtr *</td>
</tr>
<tr>
<td></td>
<td>a string representing the right side of the entry</td>
</tr>
</tbody>
</table>

Notes

The index should be between 0 and one less than the number of mapping entries.

See also

MapApi::Count()

Example

See the example for MapApi::Count().
**MapApi::GetType( int )**

Returns the type of the specified view entry.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>MapApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>int i</td>
</tr>
<tr>
<td>Returns</td>
<td>MapType</td>
</tr>
</tbody>
</table>

**Notes**

The entry type is one of MapInclude, MapExclude, and MapOverlay.

MapExclude entries negate earlier MapInclude and MapOverlay entries that map the same paths, and MapOverlay entries are not disambiguated if they overlap with earlier MapInclude entries.

In human-readable Perforce view specifications, MapExclude lines are indicated with a - character, and MapOverlay lines are indicated with a + character.

**See also**

MapApi::Count()

**Example**

See the example for MapApi::Count().
MapApi::Insert( const StrPtr & lr, MapType )

Adds a new entry to the mapping.

Virtual? No

Class MapApi

Arguments

StrPtr & lr the path to which the entry applies

MapType t the mapping type (by default, MapInclude)

Returns void

Notes

This Insert() overload is a convenience function that adds an entry with identical left and right sides. It is meant to represent mappings whose sole purpose is to include and exclude files, such as protection tables and label views.

Example

The following example demonstrates the construction and use of a protection table mapping.

```c
MapApi protect;
protect.Insert( StrRef( "//..." ) );
protect.Insert( StrRef( "//private/..." ), MapExclude );

StrBuf to;
StrBuf file1( "//depot/file.txt" );
StrBuf file2( "//private/file.txt" );

printf( "%s - access %d\n", file1.Text(), protect.Translate( file1, to ) );
printf( "%s - access %d\n", file2.Text(), protect.Translate( file2, to ) );
```

This produces the following output:

```
//depot/file.txt - access 1
//private/file.txt - access 0
```
MapApi::Insert( const StrPtr & l, const StrPtr & r, MapType t )

Adds a new entry to the mapping.

Virtual? No

Class MapApi

Arguments StrPtr & l the left side of the entry

StrPtr & r the right side of the entry

MapType t the mapping type (by default, MapInclude)

Returns void

Notes

Insert() adds one new entry to a mapping at the "bottom" (highest precedence) position in the map. The MapType parameter indicates whether the entry is a standard inclusion (the default), an exclusion, or an overlay mapping (only useful when modeling a client view).

Example

The following example demonstrates the construction and use of a branch view mapping.

```c
MapApi branch;
branch.Insert( StrRef( "//depot/main/..." ), StrRef( "//depot/rel1/..." ) );

StrBuf source( "//depot/main/file.c" );
StrBuf target;

branch.Translate( source, target );
printf( "%s -> %s\n", source.Text(), target.Text() );
```

This produces the following output:

```c
//depot/main/file.c -> //depot/rel1/file.c
```
MapApi::Join( MapApi *, MapApi * ) [static]

Joins two MapApis together to produce a combined mapping.

**Virtual?** No

**Class** MapApi

**Arguments**
- MapApi *left the first mapping
- MapApi *right the second mapping

**Returns** MapApi * a new MapApi representing the joined maps

**Notes**

This overload of Join() links the right side of the first mapping to the left side of the second mapping, as if the two mappings were laid out left to right and glued together in the middle. The resulting MapApi's left side corresponds to the first mapping's left side, and its right side corresponds to the second mapping's right side.

If the right side of the first mapping does not have anything in common with the left side of the second mapping, the resulting map is empty.

The other Join() overload allows more control over which side of each mapping is joined to the other, and the direction of the resulting mapping.

This function allocates a new MapApi object on the heap; the caller is responsible for deleting it.

**Example**

The following example demonstrates a join between a branch view and a client view.

```c
MapApi branchmap;
branchmap.Insert( StrRef( "//depot/main/..." ), StrRef( "//depot/rel1/..." ) );

MapApi clientmap;
clientmap.Insert( StrRef( "// depot/..." ), StrRef( "// client/depot/..." ) );

MapApi *branch_to_client = MapApi::Join( &branchmap, &clientmap );

StrBuf source( "//depot/main/file.c" );
StrBuf target;

branch_to_client->Translate( source, target );
printf( "%s -> %s\n", source.Text(), target.Text() );
delete branch_to_client;
```

This produces the following output:

```
//depot/main/file.c -> //client/depot/rel1/file.c
```
MapApi::Join( MapApi *, MapDir, MapApi *, MapDir ) [static]

Joins two MapApi s together to produce a combined mapping.

Virtual? No

<table>
<thead>
<tr>
<th>Class</th>
<th>MapApi</th>
</tr>
</thead>
</table>
| Arguments | MapApi *m1 the first mapping  
|          | MapDir d1 the orientation of the first mapping |
|          | MapApi *m2 the second mapping |
|          | MapDir d2 the orientation of the second mapping |
| Returns | MapApi * a new MapApi representing the joined maps |

Notes

This overload of Join() works exactly like the simpler two-argument overload, but allows the caller to reverse either or both mappings before they are joined together. Specifying MapLeftRight as the direction for both mappings will produce the same result as the two-argument Join().

If the two mappings do not have anything in common at the join point, the result is an empty mapping.

This function allocates a new MapApi object on the heap; the caller is responsible for deleting it.

Example

The following example demonstrates a join between a branch view and a client view, with both mappings reversed so that the client path is on the left side of the result and the branch source is on the right side.

```c
MapApi branchmap;
branchmap.Insert( StrRef( "//depot/main/..." ), StrRef( "//depot/rel1/..." ) );

MapApi clientmap;
clientmap.Insert( StrRef( "//depot/..." ), StrRef( "//client/depot/..." ) );

MapApi *client_to_branch = MapApi::Join
( &clientmap, MapRightLeft, &branchmap, MapRightLeft );

StrBuf clientfile( "//client/depot/rel1/file.c" );
StrBuf branchfile;

client_to_branch->Translate( clientfile, branchfile );
printf( "%s -> %s\n", clientfile.Text(), branchfile.Text() );
delete client_to_branch;
```

Executing the preceding code produces the following output:
//client/depot/rel1/file.c -> //depot/main/file.c
MapApi::Translate( const StrPtr &, StrBuf&, MapDir )

Translates a file path from one side of a mapping to the other.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>MapApi</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr &amp; from</td>
</tr>
<tr>
<td></td>
<td>StrBuf &amp; to</td>
</tr>
<tr>
<td></td>
<td>MapDir d</td>
</tr>
<tr>
<td>Returns</td>
<td>bool</td>
</tr>
</tbody>
</table>

The `Translate()` function is used to determine the effect of the mapping on any particular file. In the case of a two-sided mapping (such as a client view), it indicates where any given depot file maps in the client, or vice versa. In the case of a one-sided mapping (such as a protection table), it simply indicates whether a particular file is mapped at all.

If the specified `MapDir` is `MapLeftRight`, the input path is translated from the left side of the mapping to the right side of the mapping. If the `MapDir` is `MapRightLeft`, the mapping is effectively inverted, so that the input path is translated from the right side to the left.

If the input path does not match anything in the left side of the mapping (or the right side in the `MapRightLeft` case), the translation fails just as if the input path had been excluded from the mapping.

`Translate()` is designed to map single files. To model the effect of passing a broader path through a mapping, create a new one-sided mapping that represents that path and `Join()` it with the other mapping.

**Examples**

See the examples for `Insert()` and `Join()`.
Options methods

Options::GetValue( char opt, int subopt )

Returns the value of a flag previously stored by Options::Parse().

Virtual? No

Class Options

Arguments

<table>
<thead>
<tr>
<th>char opt</th>
<th>The flag to check</th>
</tr>
</thead>
<tbody>
<tr>
<td>int subopt</td>
<td>Return the argument associated with the subopt-th occurrence of the opt flag on the command line.</td>
</tr>
</tbody>
</table>

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:

```bash
$ 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
```
```c
#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:
%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.
```c
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>Options</code></td>
</tr>
<tr>
<td>Arguments</td>
<td>int &amp;argc</td>
</tr>
<tr>
<td></td>
<td>char ** &amp;argv</td>
</tr>
<tr>
<td></td>
<td>const char *opts</td>
</tr>
<tr>
<td></td>
<td>int flag</td>
</tr>
<tr>
<td></td>
<td>const ErrorId &amp;usage</td>
</tr>
<tr>
<td></td>
<td>Error *e</td>
</tr>
</tbody>
</table>

**Notes**

You must bypass `argv[0]` (that is, the name of the calling program) before calling `Options::Parse()`. This is most easily done by decrementing `argc` and incrementing `argv`.

An argument by be of the form `-avalue` or `-avalue`. Although an argument of the form `-avalue` is passed as two entries in `argv`, the `Options::Parse()` method parses it as one logical argument.

As arguments are scanned from the caller’s `argv`, the caller’s `argc` and `argv` are modified to reflect the arguments scanned. Scanning stops when the next argument either:

- does not begin with a `-`, or
- is a `-` only, or
- is not in the array of expected options.

Once scanning has stopped, `argc` and `argv` are returned “as-is”; that is, they are returned as they were when scanning stopped. There is no “shuffling” of arguments.

The `opts` argument is a format string indicating which options are to be scanned, and whether these options are to have associated values supplied by the user. Flags with associated values must be followed by a colon (":") or a period (".") in the format string. Using a colon allows arguments to be specified in the form `-avalue` or `-avalue`; using a period allows only the `-avalue` form.

If, based on the expectation set in the format string, the actual option string in `argv` does not provide a value where one is expected, an error is generated.
For instance, the `p4` Command Line Client’s `-V` and `-?` flags are expected to be supplied without values, but the `-p` flag is expected to be accompanied with a setting for `P4PORT`. This is the format string used by the `p4` Command Line Client:

```
```

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`):

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT_ONE</td>
<td>0x01</td>
<td>Exactly one argument is expected to remain after parsing</td>
</tr>
<tr>
<td>OPT_TWO</td>
<td>0x02</td>
<td>Exactly two arguments are expected to remain after parsing</td>
</tr>
<tr>
<td>OPT_THREE</td>
<td>0x04</td>
<td>Exactly three arguments are expected to remain after parsing</td>
</tr>
<tr>
<td>OPT_MORE</td>
<td>0x08</td>
<td>More than two arguments (three or more) are to remain after parsing</td>
</tr>
<tr>
<td>OPT_NONE</td>
<td>0x10</td>
<td>Require that zero arguments remain after parsing; if arguments remain after parsing, set an error.</td>
</tr>
<tr>
<td>OPT_MAKEONE</td>
<td>0x20</td>
<td>If no arguments remain after parsing, create one that points to <code>NULL</code>.</td>
</tr>
<tr>
<td>OPT_OPT</td>
<td>0x11</td>
<td><code>NONE</code>, or <code>ONE</code>.</td>
</tr>
<tr>
<td>OPT_ANY</td>
<td>0x1F</td>
<td><code>ONE</code>, <code>TWO</code>, <code>THREE</code>, <code>MORE</code>, or <code>NONE</code>.</td>
</tr>
<tr>
<td>OPT_DEFAULT</td>
<td>0x2F</td>
<td><code>ONE</code>, <code>TWO</code>, <code>THREE</code>, <code>MORE</code>, or <code>MAKEONE</code>.</td>
</tr>
<tr>
<td>OPT_SOME</td>
<td>0x0F</td>
<td><code>ONE</code>, <code>TWO</code>, <code>THREE</code>, or <code>MORE</code>.</td>
</tr>
</tbody>
</table>

See also

- `Options::GetValue()`
- `Options::operator[]( )`

Example

The following code and examples illustrate how `Options::Parse()` works.

```c
#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++;
            }
        }
    }
}
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:
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.

```plaintext
$ 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

$ parsedemo 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

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 one
```

```plaintext
After Options::Parse call:
  argc is 1
  argv[ 0 ] is one
```
Options::Parse( int &, StrPtr *& , const char *, int, const ErrorId &, Error * )

Extract command line arguments and associated values.

Virtual? No

Class Options

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int &amp;argc</td>
<td>Number of arguments</td>
</tr>
<tr>
<td>StrPtr * &amp;argv</td>
<td>An array of arguments to parse</td>
</tr>
<tr>
<td>const char *opts</td>
<td>The list of valid options to extract</td>
</tr>
<tr>
<td>int flag</td>
<td>A flag indicating how many arguments are expected to remain when parsing is complete</td>
</tr>
<tr>
<td>const ErrorId &amp;usage</td>
<td>An error message containing usage tips</td>
</tr>
<tr>
<td>Error *e</td>
<td>The Error object to collect any errors encountered</td>
</tr>
</tbody>
</table>

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()**.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Signaler</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**

```c
#include <unistd.h>  // for sleep()
#include <stdio.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**

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Signaler</td>
</tr>
<tr>
<td>Arguments</td>
<td>void *ptr</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**See also**

- `Signaler::OnIntr()`
- `Signaler::Intr()`
Example

```c
#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().

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Signaler</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

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

```c
#include <unistd.h> // for sleep()
#include <signal.h>
#include <stdio.h>
#include <string.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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Signaler</td>
</tr>
<tr>
<td>Arguments</td>
<td>SignalFunc callback Pointer to a function to call on receipt of an interrupt signal. The function must have the prototype voidfunc( void *ptr )</td>
</tr>
<tr>
<td></td>
<td>void *ptr Pointer to a data item to pass to the callback function when invoking it.</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

Functions are called in the reverse order that they are registered.

**See also**

- Signaler::DeleteOnIntr()
- Signaler::Intr()
Example

```c
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>Signaler</code></td>
</tr>
<tr>
<td>Arguments</td>
<td>N/A</td>
</tr>
<tr>
<td>Returns</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**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

```cpp
#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 << "\n" << 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";

    p = sb.Alloc( 70 );   // allocate in StrBuf

    cout << "sb.Text() after sb.Alloc( 70 ) returns (first three bytes) ";
    cout << "\n" << 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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>StrBuf</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>const char *buf</code> pointer to the first byte of the null-terminated string</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

**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**

```cpp
text!
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";

    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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrBuf</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *buf</td>
</tr>
<tr>
<td></td>
<td>int len</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

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

```c
#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"
    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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrBuf</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr *s  pointer to the StrPtr instance</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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

```c
#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 << "\n" << sba.Text() << "\n";
    cout << "sba.Length() after sba.Set( "xyz\" ) returns ";
    cout << sba.Length() << "\n";
    cout << "sbb.Text() after sbb.Set( "xyz\" ) returns ";
    cout << "\n" << sbb.Text() << "\n";
    cout << "sbb.Length() after sbb.Set( "xyz\" ) returns ";
    cout << sbb.Length() << "\n";

    sba.Append( sp ); // append StrPtr * to StrBuf
    cout << "sba.Text() after sba.Append( sp ) returns ";
    cout << "\n" << sba.Text() << "\n";
    cout << "sba.Length() after sba.Append( sp ) returns ";
    cout << sba.Length() << "\n";

    sbb.Append( &sr ); // append &StrRef to StrBuf
    cout << "sbb.Text() after sbb.Append( &sr ) returns ";
    cout << "\n" << sbb.Text() << "\n";
    cout << "sbb.Length() after sbb.Append( &sr ) returns ";
    cout << sbb.Length() << "\n";

    sba.Append( &sbb ); // append &StrBuf to StrBuf
    cout << "sba.Text() after sba.Append( &sbb ) returns ";
    cout << "\n" << 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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrBuf</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**

```cpp
#include <iostream>
#include <stdio.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";
    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";
    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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrBuf</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Notes**

The StrBuf constructor initializes the StrBuf to contain a zero-length null buffer.

**Example**

```cpp
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

<table>
<thead>
<tr>
<th>Class</th>
<th>StrBuf</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
<th>const StrBuf &amp;</th>
<th>(implied) reference of the StrBuf from which copying occurs</th>
</tr>
</thead>
</table>

| 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

```cpp
#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

```cpp
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><em>StrBuf</em></td>
</tr>
<tr>
<td>Arguments</td>
<td>char c the byte copied to the extended string</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**

```cpp
#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 << "\n" << sb.Text() << "\n" ;
    cout << "sb.Length() prior to sb.Extend( 'z' ) returns " ;
    cout << sb.Length() << "\n" ;
    sb.Extend( 'z' );   // extend StrBuf from char
    sb.Terminate();
    cout << "sb.Text() after sb.Extend( 'z' ) returns " ;
    cout << "\n" << 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

```cpp
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 << "\n" << sb.Text() << "\n";
    cout << "sb.Length() prior to sb.Extend( chars, 2 ) returns ",
    cout << sb.Length() << "\n";

    sb.Extend( chars, 2 ); // extend StrBuf from len bytes of char *
    sb.Terminate();

    cout << "sb.Text() after sb.Extend( chars, 2 ) returns ",
    cout << "\n" << 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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>StrBuf</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *buf (implied) pointer to the first byte of the null-terminated string</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**

```cpp
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrBuf</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrBuf &amp;buf (implied) reference of the StrBuf from which assignment occurs</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

Initialize both StrBufs 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**

```cpp
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrBuf</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr &amp;s (implied) reference of the StrPtr instance</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**

```cpp
#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

```c
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrBuf</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *s (implied) pointer to the first byte of the null-terminated string</td>
</tr>
<tr>
<td>Returns</td>
<td>StrBuf &amp; reference of the StrBuf</td>
</tr>
</tbody>
</table>

**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**

```cpp
#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";
    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 << "\n" << sb.Text() << "\n";
    cout << "sb.Length() prior to sb << i returns ";
    cout << sb.Length() << "\n";

    sb << i; // append (formatted) int to StrBuf

    cout << "sb.Text() after sb << i returns ";
    cout << "\n" << 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

```cpp
#include <iostream>
#include <stdlib.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

```cpp
#include <iostream>
#include <stdhdrs.h>
#include <strbufs.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

```c
#include <iostream>
#include <stdlib.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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrBuf</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *buf, length of the string</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**

```cpp
#include <iostream>
#include <stddef.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 << endl;
    cout << "sb.Text() returns " << sb.Text() << endl;
}
```

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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrBuf</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr *s pointer to the StrPtr instance</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**

```cpp
#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

<table>
<thead>
<tr>
<th>Class</th>
<th>StrBuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>const StrPtr &amp; s</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

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";

    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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrBuf</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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:

```cpp
sb1 = StrRef::Null();
sb2.Clear();
sb2.Terminate();
sb3.Set(" ");
sb4 = "";
```

**See also**

- `StrBuf::Clear()`
- `StrBuf::Set()`
- `StrBuf::Terminate()`
- `StrBuf::operator = (char *)`
- `StrRef::Null()`
Example

```c
#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:
<table>
<thead>
<tr>
<th>StrBuf 1</th>
<th>contains &quot;73&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>StrBuf 2</td>
<td>contains &quot;74&quot;</td>
</tr>
<tr>
<td>StrBuf 3</td>
<td>contains &quot;75&quot;</td>
</tr>
<tr>
<td>StrBuf 4</td>
<td>contains &quot;76&quot;</td>
</tr>
<tr>
<td>StrBuf 5</td>
<td>contains &quot;77&quot;</td>
</tr>
</tbody>
</table>
**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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>StrBuf</code></td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

**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:

```c
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

```c
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrDict</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr &amp;var the name of the variable to look up</td>
</tr>
<tr>
<td>Returns</td>
<td>StrPtr * the value, or NULL if not defined</td>
</tr>
</tbody>
</table>

**Notes**

For the most part, all of the following methods are equivalent:

- StrDict::GetVar( const StrPtr & )
- StrDict::GetVar( const char * )
- StrDict::GetVar( const char *, Error * )
- StrDict::GetVar( const StrPtr &, int )
- StrDict::GetVar( const StrPtr &, int, int )
- StrDict::GetVar( int, StrPtr &, StrPtr & )

The var argument must specify the name of a variable in the StrDict that you're trying to look up. In some instances, variables in a StrDict are named according to the convention FOOx or FOOx,y - one example is the tagged output of p4 filelog. Calling GetVar() with these numbers as arguments saves you the work of manually constructing the variable name by using itoa() and Append().

The version of GetVar() that returns an int is useful for iterating through a StrDict; the int argument is an index into the StrDict, and the two StrPtr arguments are set to contain the variable and value found at that index, if any. This method returns zero if there was no variable at the specified index.

**Example**

The implementation of ClientUser::OutputStat() in clientuser.cc provides a good source example:
void ClientUser::OutputStat( StrDict *varList )
{
    int i;
    StrBuf msg;
    StrRef var, val;

    // Dump out the variables, using the GetVar( x ) interface.
    // Don’t display the function, which is only relevant to rpc.
    for ( i = 0; varList->GetVar( i, var, val ); i++ )
    {
        if ( var == "func" ) continue;

        // otherAction and otherOpen go at level 2, as per 99.1 + earlier
        msg.Clear();
        msg << var << " " << val;
        char level = strncmp( var.Text(), "other", 5 ) ? '1' : '2';
        OutputInfo( level, msg.Text() );
    }

    // blank line
    OutputInfo( '0', "" );
}  

An example of output:

% p4 -Ztag filelog file.c
...
... depotFile //depot/depot/source/file.c
... rev0 3
... change0 1949
... action0 integrate
... type0 text
... time0 1017363022
... user0 testuser
... client0 testuser-luey
... desc0 <enter description here>
... how0,0 ignored
... file0,0 //depot/depot/source/old.c
... srev0,0 #1
... erev0,0 #2
... how0,1 ignored
...
...
StrDict::GetVar( const char * )

Return the value of the specified variable, or NULL if not defined.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrDict</td>
</tr>
<tr>
<td>Arguments</td>
<td>const char *var</td>
</tr>
<tr>
<td>Returns</td>
<td>StrPtr *</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>StrDict</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td>*<em>const char <em>var</em></em></td>
</tr>
<tr>
<td></td>
<td>*<em>Error <em>e</em></em></td>
</tr>
<tr>
<td>Returns</td>
<td>**StrPtr ***</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Class</th>
<th>StrDict</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
<th>const StrPtr &amp;var</th>
<th>the name of the variable to look up</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>StrPtr *</th>
<th>the value, or NULL if not defined</th>
</tr>
</thead>
</table>

Notes

For the most part, all of the GetVar() methods are equivalent.

For details, see StrDict::GetVar( const StrPtr & )
### StrDict::GetVar( const StrPtr &var, int, int )

Return the value of the specified variable, or NULL if not defined.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrDict</td>
</tr>
<tr>
<td>Arguments</td>
<td><code>const StrPtr &amp;var</code> the name of the variable to look up</td>
</tr>
<tr>
<td></td>
<td><code>int x</code> appended to the variable's name</td>
</tr>
<tr>
<td></td>
<td><code>int y</code> appended to the variable's name</td>
</tr>
<tr>
<td>Returns</td>
<td><code>StrPtr *</code> the value, or NULL if not defined</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrDict</td>
</tr>
<tr>
<td>Arguments</td>
<td>int i</td>
</tr>
<tr>
<td></td>
<td>StrPtr &amp;var</td>
</tr>
<tr>
<td></td>
<td>StrPtr &amp;val</td>
</tr>
<tr>
<td>Returns</td>
<td>int</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Class</th>
<th>StrDict</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
<th>FILE *i</th>
<th>the file to load from</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Returns</th>
<th>int</th>
<th>always equals 1</th>
</tr>
</thead>
</table>

**Notes**

`Load()` loads a *StrDict* from a file previously created by `Save()`.

**Example**

The following example "loads" a *StrDict* by reading it from `stdin`.

```c
MyStrDict sd;
ClientUser ui;

sd.Load( stdin );
ui.OutputStat( &sd );
```

Given a marshaled *StrDict* on `stdin`, the code produces the following output:

```bash
> cat marshaled.strdict

depotFile=/depot/file.c
clientFile=c:\test\depot\file.c
headAction=edit
headType=text
headTime=1020067607
headRev=4
headChange=2042
headModTime 1020067484
func=client-FstatInfo

> a.out < marshaled.strdict

... depotFile //depot/file.c
... clientFile clientFile=c:\test\depot\file.c
... headAction edit
... headType text
... headTime 1020067607
... headRev 4
... headChange 2042
... headModTime 1020067484
```
**StrDict::Save( FILE * )**

Marshals the StrDict into a text file.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrDict</td>
</tr>
<tr>
<td>Arguments</td>
<td>FILE *out the file to save to</td>
</tr>
<tr>
<td>Returns</td>
<td>int always equals 1</td>
</tr>
</tbody>
</table>

**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.

```c
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
cclientFile=c:\test\depot\file.c
headAction=edit
headType=text
headTime=1020067607
headRev=4
headChange=2042
headModTime=1020067484
func=client-FstatInfo
```
**StrDict::SetArgv( int, char *const *)**

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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td><strong>StrDict</strong></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>int argc</td>
</tr>
<tr>
<td></td>
<td>char *const *argv</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

SetArgv() is typically used when setting command arguments in ClientApi.

**Example**

p4api.cc provides an example of using SetArgv() to set arguments.

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrNum</td>
</tr>
<tr>
<td>Arguments</td>
<td>int v  the number to store (optional)</td>
</tr>
<tr>
<td>Returns</td>
<td>StrNum</td>
</tr>
</tbody>
</table>

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:

```cpp
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrNum</td>
</tr>
<tr>
<td>Arguments</td>
<td>int v the number to store</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

A StrNum always stores numbers using base ten.

**Example**

```cpp
#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

```c
#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**

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td><strong>StrOps</strong></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>StrPtr &amp;o</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

Unprintable characters are displayed as hexadecimal ASCII values, surrounded by greater-than/less-than characters.

**Example**

```c
#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:

```cpp
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 &o, const StrPtr &s )

Make a copy of a string, with each line indented.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrOps</td>
</tr>
<tr>
<td>Arguments</td>
<td>StrBuf &amp;o the output string</td>
</tr>
<tr>
<td></td>
<td>StrPtr &amp;s the input string</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

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

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrOps</td>
</tr>
<tr>
<td>Arguments</td>
<td>StrBuf &amp;o the input string</td>
</tr>
<tr>
<td></td>
<td>char *vec[] the output array</td>
</tr>
<tr>
<td></td>
<td>int maxVec the maximum number of lines to handle</td>
</tr>
<tr>
<td>Returns</td>
<td>int the actual number of lines handled</td>
</tr>
</tbody>
</table>

**Notes**

This function handles all types of line breaks: "\x", "\n", and "\r\n".

**Example**

```c
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

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrOps</td>
</tr>
<tr>
<td>Arguments</td>
<td>StrBuf &amp;o the string to convert to lowercase</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

This function modifies an original string in place by converting all uppercase characters to lowercase.

**Example**

```c
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

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *octet</td>
<td>the input stream</td>
</tr>
<tr>
<td>int len</td>
<td>length of the input in bytes</td>
</tr>
<tr>
<td>StrBuf &amp;x</td>
<td>the output string</td>
</tr>
</tbody>
</table>

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

```c
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

<table>
<thead>
<tr>
<th>Class</th>
<th>StrOps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>StrBuf &amp;o  the output string</td>
</tr>
<tr>
<td></td>
<td>StrPtr &amp;i  the input string</td>
</tr>
<tr>
<td></td>
<td>StrBuf &amp;s  the substring to match</td>
</tr>
<tr>
<td></td>
<td>StrPtr &amp;r  the substring to replace s</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

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

```c
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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrOps</td>
</tr>
<tr>
<td>Arguments</td>
<td>StrPtr &amp;string the string on which to operate</td>
</tr>
<tr>
<td></td>
<td>target the target character</td>
</tr>
<tr>
<td></td>
<td>replace the character with which to replace target</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**

```c
#include <stdhdrs.h>
#include <strbuf.h>
#include <strops.h>

int main( int argc, char **argv )
{
    StrBuf sb;
    sb.Set( "\tPassword" );
    StrOps::Sub( sb, 'o', '0' );
    StrOps::Sub( sb, 'a', '4' );
    StrOps::Dump( sb );
    return 0;
}
```

Executing the preceding code produces the following output:

P4ssw0rd
**StrOps::Upper( StrBuf & )**

Convert each character in a string (in place) to uppercase

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>StrOps</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>StrBuf &amp;o</code>  the string to convert to uppercase</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

**Notes**

This function modifies an original string in place by converting all lowercase characters to uppercase.

**Example**

```c
StrBuf o = StrBuf();
o.Set( "xYzZy" );

printf( "Before: %s\n", o );
StrOps::Upper( o );
printf( "After:  %s\n", o );

return 0;
```

Executing the preceding code produces the following output:

```
% a.out
Before: xYzZy
After: XYZZY
```
StrOps::Words( StrBuf &, const char *[], char *[], int )

Break a string apart at whitespace.

**Virtual?** No

**Class** StrOps

**Arguments**
- StrBuf &tmp a temporary string
- const char *buf the input string
- char *vec[] the output array
- int maxVec the maximum number of words to handle

**Returns** int the actual number of words handled

**Notes**
This function uses the isAspace() function to define whitespace.

**Example**

```c
StrBuf o = StrBuf();
StrBuf tmp = StrBuf();
o.Set( "abc\tdef \nghi\nxyz xyzzy plugh" );

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

char *vec[5];
int w = StrOps::Words( tmp, o, vec, 5 );

for ( ; w ; w-- )
{
    printf( "Word %d: %s\n", w, vec[w-1] );
}
return 0;
```

Executing the preceding code produces the following output:

```
Input StrBuf:
abc    def   ghi
xyz xyzzy plugh

Word 5: xyzzy
Word 4: xyz
Word 3: ghi
Word 2: def
Word 1: abc
```
**StrOps::XtoO( char *, unsigned char *, int )**

Convert a hex string into an octet stream.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrOps</td>
</tr>
<tr>
<td>Arguments</td>
<td>char *x</td>
</tr>
<tr>
<td></td>
<td>char *octet</td>
</tr>
<tr>
<td></td>
<td>int octlen</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

This function converts the input hexadecimal string into the stream of bytes that it represents.

**Example**

```c
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
\section*{StrPtr methods}

\subsection*{StrPtr::Atoi()}

Return the numeric value, if any, represented by this StrPtr's buffer.

\begin{itemize}
\item \textbf{Virtual?}: No
\item \textbf{Class}: StrPtr
\item \textbf{Arguments}: None
\item \textbf{Returns}: int integer value of the string
\end{itemize}

\textbf{Notes}

StrPtr::Atoi() is equivalent to calling atoi(StrPtr::Text()). Non-numeric strings typically return a value of zero.

\textbf{Example}

\begin{verbatim}
#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() );
}
\end{verbatim}

Executing the preceding code produces the following output:

\begin{verbatim}
123 + 234 = 357
\end{verbatim}
**StrPtr::CCompare( const StrPtr & )**

Case insensitive comparison of two StrPtrs.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrPtr</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr &amp;s the StrPtr to compare this one with</td>
</tr>
<tr>
<td>Returns</td>
<td>int zero if identical, nonzero if different</td>
</tr>
</tbody>
</table>

**Notes**

StrPtr::CCompare() is a wrapper for strcmp() 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**

```c
#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 StrPtrs, with case sensitivity based on client platform.

Virtual?  No

Class  StrPtr

Arguments  const StrPtr &s  the StrPtr to compare this one with

Returns  int  zero if identical, nonzero if different

Notes

StrPtr::Compare() is a wrapper for zstrcmp(). Its return value, if nonzero, indicates which of the two strings is "greater" in the ASCII sense.

See also

StrPtr::CCompare()
StrPtr::XCompare().

Example

```c
#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
```
abc != Abc
abc != xyz
**StrPtr::Contains( const StrPtr & )**

Look for a substring and, if found, return it.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><strong>StrPtr</strong></td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr &amp;s the substring to look for</td>
</tr>
<tr>
<td>Returns</td>
<td>char * the start of the substring if found, otherwise NULL</td>
</tr>
</tbody>
</table>

**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**

```c
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1, str2;

    str1.Set( "the quick brown fox jumps over the lazy dog" );
    str2.Set( "brown fox" );

    printf( "%s\n", str1.Contains( str2 ) );
    return 0;
}
```

Executing the preceding code produces the following output:

brown fox jumps over the lazy dog
**StrPtr::Length()**

Return the length of this StrPtr.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td><strong>StrPtr</strong></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>int</td>
</tr>
</tbody>
</table>

**Example**

```c
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrPtr</td>
</tr>
<tr>
<td>Arguments</td>
<td>int x the index to look in</td>
</tr>
<tr>
<td>Returns</td>
<td>char the character at that index</td>
</tr>
</tbody>
</table>

**Notes**

This operator does no bounds checking, and can therefore return data from beyond the end of the string.

**Example**

```c
#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\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 *`.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td><code>StrPtr</code></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td><code>const char *buf</code> the string to compare with</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td><code>int</code> zero if the comparison is false, nonzero if true.</td>
</tr>
</tbody>
</table>

**Notes**

These operators are typically used in simple comparisons between `StrPtrs`, such as to see whether two `StrPtrs` contain the same string, or whether one is greater than the other, ASCII-wise. The comparison is always case-sensitive.

**Example**

```c
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;
    str1.Set( "This string" );
    printf( "%s", str1.Text());
    if ( str1 == "that string" ) printf( " == ");
    if ( str1 > "that string" ) printf( " > ");
    if ( str1 < "that string" ) printf( " < ");
    printf( "that string" );
    return 0;
}
```

Executing the preceding code produces the following output:

```
This string < that string
```

(Note that "t" > "T" in ASCII.)
**StrPtr::operators ==, !==, >, <, <=, >= ( const StrPtr & )**

Case-sensitive comparison operators between StrPtr and StrPtr.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrPtr</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr &amp; buf</td>
</tr>
<tr>
<td>Returns</td>
<td>int</td>
</tr>
</tbody>
</table>

**Notes**

These operators are typically used in simple comparisons between StrPtrs, such as to see whether two StrPtrs contain the same string, or whether one is greater than the other, ASCII-wise. The comparison is always case-sensitive.

**Example**

```c
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1, str2;

    str1.Set( "This string" );
    str2.Set( "that string" );

    printf( "%s", str1.Text() );
    if ( str1 == str2 ) printf( " == " );
    if ( str1 > str2 ) printf( " > " );
    if ( str1 < str2 ) printf( " < " );
    printf( "%s\n", str2.Text() );
    return 0;
}
```

Executing the preceding code produces the following output:

```
This string < that string
```

(Note that "t" > 'T' in ASCII.)
StrPtr::Text()

Return the char * containing this StrPtr's text.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrPtr</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>char *</td>
</tr>
</tbody>
</table>

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

```c
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;
    str1.Set( "the quick brown fox jumps over the lazy dog" );
    printf( "%s\n", str1.Text() );
    return 0;
}
```

Executing the preceding code produces the following output:

the quick brown fox jumps over the lazy dog
**StrPtr::Value()**

Return the char * containing this StrPtr’s text.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrPtr</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>char *</td>
</tr>
</tbody>
</table>

**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**

```c
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    StrBuf str1;
    str1.Set( "the quick brown fox jumps over the lazy dog" );
    printf( "%s
", 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 *StrPtrs*.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td><em>StrPtr</em></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>const <em>StrPtr</em> &amp;s the <em>StrPtr</em> to compare this one with</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>int zero if identical, nonzero if different</td>
</tr>
</tbody>
</table>

**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**

```c
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
<td><code>StrRef</code></td>
</tr>
<tr>
<td><strong>Arguments</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td><code>StrRef</code></td>
</tr>
</tbody>
</table>

**Notes**

If arguments are provided, the constructor calls `set()` with them.
**StrRef::StrRef( const StrPtr & ) (constructor)**

Construct a StrRef, referencing an existing string.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrRef</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr &amp; a StrPtr to reference</td>
</tr>
<tr>
<td>Returns</td>
<td>StrRef</td>
</tr>
</tbody>
</table>

**Notes**

If arguments are provided, the constructor calls *set()* with them.

**Example**

```cpp
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>StrRef</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>char *buf</code> a null-terminated string to reference</td>
</tr>
<tr>
<td>Returns</td>
<td><code>StrRef</code></td>
</tr>
</tbody>
</table>

**Notes**

If arguments are provided, the constructor calls `Set()` with them.

**Example**

```c
#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:

```plaintext
chars[] = "abc"
sr.Text() returns "abc"
```
StrRef::StrRef( const char * , int ) (constructor)

Construct a StrRef, referencing an existing string.

Virtual? No

Class StrRef

Arguments

<table>
<thead>
<tr>
<th>char *buf</th>
<th>a null-terminated string to reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>int len</td>
<td>the string length</td>
</tr>
</tbody>
</table>

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

```c++
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrRef</td>
</tr>
<tr>
<td>Arguments</td>
<td>None</td>
</tr>
<tr>
<td>Returns</td>
<td>StrPtr an empty StrPtr</td>
</tr>
</tbody>
</table>

**Notes**

*StrRef::Null()* is a static function.

**Example**

```cpp
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrRef</td>
</tr>
<tr>
<td>Arguments</td>
<td>StrPtr &amp;s</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**Notes**

The = operator is equivalent to calling `Set()`.

**Example**

```cpp
#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

```cpp
#include <iostream>
#include <stdhdrs.h>
#include <strbuf.h>

int main( int argc, char **argv )
{
    char chars[] = "xyz";
    StrRef sr;
    sr = chars;
    sr = chars;
    cout << "chars[] = " << chars << endl;    
    cout << "sr.Text() returns " << sr.Text() << endl;
}
```

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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td><code>StrRef</code></td>
</tr>
<tr>
<td>Arguments</td>
<td><code>int len</code> the amount by which to move the pointer</td>
</tr>
<tr>
<td>Returns</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

#### 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

```c++
#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" << endl;
    cout << "sr.Text() returns \"" << sr.Text() << "\n" << endl;
}
```

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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrRef</td>
</tr>
<tr>
<td>Arguments</td>
<td>char *buf the null-terminated string to reference</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**

```c++
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrRef</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
<th>char *buf</th>
<th>the null-terminated string to reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int len</td>
<td>the length of the string</td>
</tr>
</tbody>
</table>

| 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**

```c
#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.

<table>
<thead>
<tr>
<th>Virtual?</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>StrRef</td>
</tr>
<tr>
<td>Arguments</td>
<td>const StrPtr *s the value to set</td>
</tr>
<tr>
<td>Returns</td>
<td>void</td>
</tr>
</tbody>
</table>

**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**

```cpp
#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**

```cpp
#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"
```
Appendix

License Statements

Perforce software includes software developed by the University of California, Berkeley and its contributors. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

Perforce software includes software from the Apache ZooKeeper project, developed by the Apache Software Foundation and its contributors. (http://zookeeper.apache.org/)