PERFORCE

P4 User's Guide

2015.1

March 2015

P4 User's Guide
2015.1

March 2015

Copyright © 1999-2015 Perforce Software.

All rights reserved.

Perforce software and documentation is available from `http://www.perforce.com/`. You can download and use Perforce programs, but you can not sell or redistribute them. You can download, print, copy, edit, and redistribute the documentation, but you can not sell it, or sell any documentation derived from it. You can not modify or attempt to reverse engineer the programs.

This product is subject to U.S. export control laws and regulations including, but not limited to, the U.S. Export Administration Regulations, the International Traffic in Arms Regulation requirements, and all applicable end-use, end-user and destination restrictions. Licensee shall not permit, directly or indirectly, use of any Perforce technology in or by any U.S. embargoed country or otherwise in violation of any U.S. export control laws and regulations.

Perforce programs and documents are available from our Web site as is. No warranty or support is provided. Warranties and support, along with higher capacity servers, are sold by Perforce Software.

Perforce Software assumes no responsibility or liability for any errors or inaccuracies that might appear in this book. By downloading and using our programs and documents you agree to these terms.

Perforce and Inter-File Branching are trademarks of Perforce Software.

All other brands or product names are trademarks or registered trademarks of their respective companies or organizations.

Any additional software included within Perforce software is listed in License Statements on page 119.

Table of Contents

About This Manual

This guide tells you how to use the Perforce Command-Line Client (`p4`). If you're new to version management systems, you don't know basic Perforce concepts, or you've never used Perforce before, read *Introducing Perforce* before reading this guide. This guide assumes a good basic understanding of version control.

Command line versus GUIs

Perforce provides many applications that enable you to manage your files, including the Perforce Command-Line Client, GUIs such as P4V, and plug-ins. The Perforce Command-Line Client enables you to script and to perform administrative tasks that are not supported by Perforce GUIs.

Getting started with Perforce

If this is your first time working with Perforce, here's how to get started:

1. Read *Introducing Perforce* to learn the basics.

   At a minimum, learn the following concepts: *changelist*, *depot*, *client workspace*, *sync*, and *submit*. For short definitions, refer to the glossary at the back of this guide.

2. Ask your Perforce administrator for the host and port for your Perforce service.

   If you intend to experiment with Perforce and don't want to risk damaging your production depot, ask the Perforce administrator to start another service for test purposes. For details about installing the Perforce service, refer to the *Perforce Server Administrator's Guide: Fundamentals*.

3. Use this guide to help you install the Perforce Command-Line Client and configure your client workspace, unless your system administrator has already configured your machine. See Chapter 2, "Configuring P4" on page 3, for details.

4. Learn to perform the following tasks:

   - *sync* (transfer selected files from the repository to your computer)
   - *submit* (transfer changed files from your workspace to the repository)
   - *revert* (discard changes)

   See Chapter 4, "Managing Files and Changelists" on page 37, for details.

5. Learn to refine your client view. See "Refining workspace views" on page 10 for details.

These basic skills enable you to do much of your daily work. Other tasks involving code base maintenance (streams, branching and labeling) and workflow (jobs) tend to be less frequently done. This guide includes details about performing these tasks using `p4` commands.

Perforce documentation

This guide, the *P4 Command Reference*, and the `p4 help` command are the primary documentation for the Perforce Command-Line Client. This guide describes the current release. For documentation for older releases, refer to the Perforce web site.

For documentation on other Perforce applications, see the documentation web page at http://www.perforce.com.

For specific information about…	See this documentation
The basics of Perforce.	*Introducing Perforce*
Installing and administering the Perforce service, including user management, security settings and configuring distributed environments that include proxies, replicas, and edge servers.	*Perforce Server Administrator's Guide: Fundamentals* and *Perforce Server Administrator's Guide: Multi-site Deployment*
`p4` command line flags and options (reference).	*P4 Command Reference*, `p4 help`
P4V, the cross-platform Perforce Visual Client.	*P4V User's Guide*, P4V online help
Working offline with the Perforce Sandbox.	*Perforce Sandbox User's Guide*
Perforce plug-ins and integrations.	IDEs: *Using IDE Plug-ins* Defect trackers: *Defect Tracking Gateway Guide* Others: online help from the Perforce menu or web site
Developing custom Perforce applications using the Perforce C/C++ API.	*C/C++ API User's Guide*
Working with Perforce in Ruby, Perl, Python, and PHP.	*APIs for Scripting*

Please give us feedback

We are interested in receiving opinions on this guide from our users. In particular, we'd like to hear from users who have never used Perforce before. Does this guide teach the topic well? Please let us know what you think; we can be reached at `<manual@perforce.com>`.

What's new in this guide for 2015.1

This section provides a list of changes to this guide for the Perforce Server 2015.1 release. For a list of all new functionality and major bug fixes in Perforce Server 2015.1, see the Perforce Server 2015.1 Release Notes.

Additional commands that take a file revision range.	See "Revision ranges" on page 32 for updated information on all commands that can take a file revision range.
Locking behavior of files following a failed submit	See "Submitting changelists" on page 44 for updated information about locking behavior.
Added information on autoreload for labels	See "Using static labels to archive workspace configurations" on page 82 for updated information on storage options and performance.

Chapter 1 Installing P4

This chapter tells you how to install the Perforce Command-Line Client (p4) on your workstation. For details about installing the Perforce versioning service, refer to the *Perforce Server Administrator's Guide: Fundamentals*.

Installing P4 on UNIX and OS X

To install the Perforce Command-Line Client (p4) on a UNIX or Mac OS X machine, perform the following steps:

1. Download the p4 executable file from the Perforce web site:

   http://www.perforce.com/downloads/complete_list

   Perforce applications are typically installed into /usr/local/bin.

2. Make the p4 file executable (chmod +x p4).

3. Configure the port setting, client workspace name, and user name. You can specify these settings by configuring the P4PORT, P4CLIENT, and P4USER environment variables. (For details, see Chapter 2, "Configuring P4" on page 3.)

Installing P4 on Windows

To install the Perforce Command-Line Client (p4.exe) on Windows, download and run the Perforce Windows installer (perforce.exe) from the Downloads page of the Perforce web site:

http://www.perforce.com/downloads/complete_list

The Perforce installer enables you to install and uninstall the Perforce Command-Line Client and other Perforce Windows components.

Verifying the installation

To verify that you have successfully installed the Perforce Command-line Client, type p4 info at the command line and press ENTER. If the Perforce service is running on the specified host and port, the following message is displayed:

```
User name: ona
Client name: ona-agave
Client host: agave
Client root: /home/ona/p4-ona
Current directory: /home/ona/p4-ona
Client address: 10.0.0.196
Server address: perforce:1666
Server root: /usr/depot/p4d
Server date: 2012/03/28 12:11:47 -0700 PDT
Server uptime: 752:41:33
Server version: P4D/FREEBSD/2012.1/406375 (2012/01/25)
Server license: P4Admin <p4adm> 20 users (expires 2013/01/01)
Server license-ip: 10.0.0.2
Case handling: sensitive
```

If your configuration settings are incorrect, an error message is displayed:

```
Perforce client error:
        Connect to server failed; check $P4PORT.
        TCP connect to <hostname> failed.
        <hostname>: host unknown.
```

If your administrator has configured Perforce to require SSL, the first time you attempt to connect to the Perforce service, you will need to verify the server's fingerprint. See "SSL-encrypted connections" on page 19.

Chapter 2

Configuring P4

This chapter tells you how to configure connection settings.

Configuration overview

Perforce is an enterprise version management system in which you connect to a shared versioning service; users *sync* files from the shared repository, called the *depot*, and edit them on your workstation in your *client workspace*. This chapter assumes that your system administrator has configured your organization's Perforce service. For details about setting up the versioning service, refer to the *Perforce Server Administrator's Guide: Fundamentals*.

To set up your workspace so you can work with Perforce, perform the following steps:

1. Configure settings for the protocol, host, and port (so you can connect to the Perforce service). See "Configuring Perforce settings" on page 4.
2. Define your workspace (at a minimum, assign a name and specify a workspace root where you want local copies of depot files stored). See "Defining client workspaces" on page 7.
3. Verify the connection. See "Verifying connections" on page 9.

After you configure your workspace, you can populate it by syncing files that are stored in the depot. For details, see "Syncing (retrieving) files" on page 38 and the description of the `p4 sync` command in the *P4 Command Reference*.

Before you start to configure Perforce, ask your Perforce administrator for the proper host and port setting. Also ask whether a workspace has already been configured for your workstation.

What is a client workspace?

A Perforce *client workspace* is a set of directories on your workstation where you work on file revisions that are managed by Perforce. Each workspace is given a name that identifies the client workspace to the Perforce service. If no workspace name is specified (by setting the `P4CLIENT` environment variable) the default workspace name is the name of your workstation. To specify the effective workspace name, set the `P4CLIENT` environment variable. You can have multiple workspaces on your machine.

All files within a Perforce client workspace share a root directory, called the *client workspace root*. The workspace root is the highest-level directory of the workspace under which the managed source files reside.

If you configure multiple workspaces on the same machine, keep workspace locations separate to avoid inadvertently overwriting files. Ensure that client roots are located in different folders and that their workspace views do not map depot files to overlapping locations on your workstation.

After you configure your workspace, you can sync files from the depot and submit changes. For details about these tasks, refer to Chapter 4, "Managing Files and Changelists" on page 37.

How Perforce manages the workspace

Perforce manages the files in a client workspace as follows:

- Files in the workspace are created, updated, and deleted as determined by your changes.
- Write permission is enabled when you edit a file, and disabled when you submit your changes.

The state of your workspace is tracked and managed by Perforce. To avoid conflicts with the file management performed by Perforce applications, do not manually change read-only permission settings on files. Perforce has commands that help you determine whether or not the state of your client workspace corresponds to Perforce's record of that state; see "Working offline" on page 48 for details.

Files in the workspace that you have not put under Perforce control are ignored by Perforce. For example, compiled objects, libraries, executables, and developers' temporary files that are created while developing software but not added to the depot are not affected by Perforce commands.

After defining your client workspace, you can fine-tune the workspace definition. Probably most important, you can restrict the portion of the depot that is visible to you, to prevent you from inadvertently syncing the entire depot. For details, refer to "Refining workspace views" on page 10.

Configuring Perforce settings

This guide refers to Perforce settings using environment variables (for example, `set P4CLIENT`), but you can specify Perforce settings such as port, user, and workspace names using the following methods, listed in order of precedence:

1. On the command line, using options
2. In a config file, if `P4CONFIG` is set
3. User environment variables (on UNIX or Windows)
4. System environment variables (on Windows, system-wide environment variables are not necessarily the same thing as user environment variables)
5. On Windows or OS X, in the user registry or settings (set by issuing the `p4 set` command)
6. On Windows or OS X, in the system registry or system settings (set by issuing the `p4 set -s` command)

To configure your workstation to connect to the Perforce service, you specify the name of the host where the service is running, and the port on which it is listening. The default host is `perforce` and default port is 1666. If the service is running on your own machine, specify `localhost` as the host name. If the service is running on port 1666, you can omit the port specification.

You can specify these settings as described in the following sections. For details about working offline (without a connection to a Perforce service), see "Working offline" on page 48.

Using the command line

To specify these settings on the command line, use the `-p` option. For example:

```
p4 -p tcp:localhost:1776 sync //depot/dev/main/jam/Jambase
```

Settings specified on the command line override any settings specified in config files, environment variables, the Windows registry, or OS X system settings. For more details about command-line options, refer to the discussion of global options in the *P4 Command Reference*.

Using config files

Config files are text files containing Perforce settings that are in effect for files in and below the directory where the config file resides. Config files are useful if you have multiple client workspaces on the same machine. By specifying the settings in config files, you avoid the inconvenience of changing system settings every time you want to work with a different workspace.

To use config files, you define the `P4CONFIG` environment variable, specifying a file name (for example, `.p4config`). When you issue a command, Perforce searches the current working directory and its parent directories for the specified file and uses the settings it contains (unless the settings are overridden by command-line options).

Each setting in the file must be specified on its own line, using the following format:

`setting=value`

The following settings can be specified in a config file.

Setting	Description
`P4CHARSET`	Character set used for translation of Unicode files.
`P4COMMANDCHARSET`	Non-UTF-16 or UTF-32 character set used by Command-Line Client when `P4CHARSET` is set to a UTF-16 or UTF-32 character set.
`P4CLIENT`	Name of the current client workspace.
`P4DIFF`	The name and location of the diff program used by `p4 resolve` and `p4 diff`.
`P4EDITOR`	The editor invoked by those Perforce commands that use forms.
`P4HOST`	Hostname of the client workstation. Only useful if the `Host:` field of the current client workspace has been set in the `p4 client` form.
`P4IGNORE`	A list of files to ignore when using the `p4 add` and `p4 reconcile` commands.
`P4LANGUAGE`	This environment variable is reserved for system integrators.
`P4MERGE`	The name and location of the third-party merge program to be used by `p4 resolve`'s merge option.
`P4PASSWD`	Supplies the current Perforce user's password for any Perforce command.
`P4PORT`	The protocol, host and port number of the Perforce service (including proxies or brokers) with which to communicate.

Setting	Description
P4TRUST	The location of a file of known (trusted) Perforce servers. You manage the contents of this file with the `p4 trust` command. By default, this file is `.p4trust` in your home directory.
P4USER	Current Perforce user name.

For details about these settings, refer to the *P4 Command Reference*.

Example 2.1. Using config files to handle switching between two workspaces.

Ona switches between two workspaces on the same machine. The first workspace is `ona-ash`. It has a client root of `/tmp/user/ona` and connects to the Perforce service using SSL at `ssl:ida:1818`. The second workspace is called `ona-agave`. Its client root is `/home/ona/p4-ona`, and it uses a plaintext connection to a Perforce service at `tcp:warhol:1666`.

Ona sets the `P4CONFIG` environment variable to `.p4settings`. She creates a file called `.p4settings` in `/tmp/user/ona` containing the following text:

```
P4PORT=ssl:ida:1818P4CLIENT=ona-ash
```

She creates a second `.p4settings` file in `/home/ona/p4-ona`. It contains the following text:

```
P4PORT=tcp:warhol:1666P4CLIENT=ona-agave
```

Any work she does on files under `/tmp/user/ona` is managed by the Perforce service at `ssl:ida:1818` and work she does on files under `/home/ona/p4-ona` is managed by the Perforce service at `tcp:warhol:1666`.

Using environment variables

To configure connection settings using environment variables, set `P4PORT` to ***`protocol`***`:`***`host`***`:`***`port`***, as in the following examples.

If the service runs on	and listens to port	supports encryption protocol	set P4PORT to
your computer	`1666`	nothing (plaintext)	`localhost:1666`
`perforce`	`1666`	SSL	`ssl:perforce:1666`
`houston`	`3435`	nothing (plaintext)	`tcp:houston:3435`
`example.com`	`1818`	SSL	`ssl:example.com:1818`

If you do not specify a protocol in your `P4PORT` setting, `tcp:` (plaintext communication over TCP/IP) is assumed. If the Perforce service has been configured to support SSL, you can encrypt your connection to Perforce by using `ssl:` as the desired protocol.

Other protocols (for example, `tcp4:` to require a plaintext IPv4 connection, or `ssl64:` to require an encrypted connection, but to prefer the use of the IPv6 transport instead of IPv4) are available for use in mixed networking environments.

See "Connecting over IPv6 networks" on page 9, and the *Perforce Server Administrator's Guide: Fundamentals*, for details.

Using the Windows registry or OS X system settings

On Windows and OS X machines, you can store connection settings in the registry (or system settings) by using the `p4 set` command. For example:

```
p4 set P4PORT=ssl:tea.example.com:1667
```

There are two ways you can configure Perforce settings in the registry:

- `p4 set` *`setting=value`*: for the current local user.
- `p4 set -s` *`setting=value`*: for all users on the local machine. Can be overridden by any registry settings made for the local user. Requires administrative privileges.

To see which settings are in effect, use the `p4 set` command without arguments. For details about the `p4 set` command, see the *P4 Command Reference*.

Defining client workspaces

To define a client workspace:

1. Specify the workspace name by setting `P4CLIENT`; for example, on a UNIX system:

   ```
   $ P4CLIENT=bruno_ws ; export P4CLIENT
   ```

2. Issue the `p4 client` command.

   Perforce displays the client workspace specification form in your text editor. (For details about Perforce forms, refer to "Using Perforce forms" on page 34.)

3. Specify (at least the minimum) settings and save the specification.

No files are synced when you create a client specification. To find out how to sync files from the depot to your workspace, refer to "Syncing (retrieving) files" on page 38. For details about relocating files on your machine, see "Changing the location of your workspace" on page 16.

The minimum settings you must specify to configure a client workspace are:

- **Workspace name**

  The workspace name defaults to your machine's hostname, but a your workstation can contain multiple workspaces. To specify the effective workspace, set `P4CLIENT`.

- **Workspace root**

> The client workspace root is the top directory of your client workspace, where Perforce stores your working copies of depot files. Be sure to set the workspace root, or you might inadvertently sync files to your workstation's root directory. (When specifying a workspace root on Windows, you must also include the drive letter.)
>
> If the workspace root directory does not exist, you must create it before the Perforce application can make use of it.
>
> The `@`, `#`, `*`, and `%` characters have specific meaning to Perforce; if you have file or folder names that use these characters, see "Restrictions on filenames and identifiers" on page 28 for details.

Your *client workspace view* determines which files in the depot are mapped to your workspace and enables Perforce to construct a one-to-one mapping between individual depot and workspace files. You can map files to have different names and locations in your workspace than they have in the depot, but you cannot map files to multiple locations in the workspace or the depot. By default, the entire depot is mapped to your workspace. You can define a client workspace view to map only files and directories of interest, so that you do not inadvertently sync the entire depot into your workspace. For details, see "Refining workspace views" on page 10.

Example 2.2. Setting the workspace view.

Bruno issues the `p4 client` command and sees a form containing this default client workspace view definition:

```
Client:      bruno_ws
Update:      2014/05/12 09:46:53
Access:      2014/05/12 10:28:40
Owner:       bruno
Host:        dhcp_24-n102.dhcp.perforce.com
Description: Created by jbruges.
Root:        c:\bruno_ws
Options:     :noallwrite noclobber nocompress unlocked nomodtime normdir
SubmitOptions: submitunchanged
LineEnd: local
View:
        //depot/...     //bruno_ws/...
```

He modifies the view to map only the development portion of the depot.

```
View:
        //depot/dev/...     //bruno_ws/dev/...
```

He further modifies the view to map files from multiple depots into his workspace.

```
View:
        //depot/dev/...     //bruno_ws/depot/dev/...
        //testing/...       //bruno_ws/testing/...
        //archive/...       //bruno_ws/archive/...
```

Verifying connections

To verify a connection, issue the **p4 info** command. If **P4PORT** is set correctly, information like the following is displayed:

```
User name: bruno
Client name: bruno_ws
Client host: workstation_12
Client root: c:\bruno_ws
Current directory: c:\bruno_ws
Peer address; 10.0.102.24:61122
Client address: 10.0.0.196
Server address: ssl:example.com:1818
Server root: /usr/depot/p4d
Server date: 2012/03/28 15:03:05 -0700 PDT
Server uptime: 752:41:33
Server version: P4D/FREEBSD/2012.1/406375 (2012/01/25)
ServerID: Master
Server license: P4Admin <p4adm> 20 users (expires 2015/01/01)
Server license-ip: 10.0.0.2
Case handling: sensitive
```

The **Server address:** field shows the host to which **p4** connected and also displays the host and port number on which the Perforce service is listening. If **P4PORT** is set incorrectly, you receive a message like the following:

```
Perforce client error:
   Connect to server failed; check $P4PORT.
   TCP connect to perforce:1666 failed.
   perforce: host unknown.
```

If the value you see in the third line of the error message is **perforce:1666** (as above), **P4PORT** has not been set. Set **P4PORT** and try to connect again.

If your installation requires SSL, make sure your **P4PORT** is of the form **ssl:*hostname*:*port***.

You will be asked to verify the server's fingerprint the first time you attempt to connect to the service. If the fingerprint is accurate, use the **p4 trust** command to install the fingerprint into a file (pointed to by the **P4TRUST** environment variable) that holds a list of known/trusted Perforce servers and their respective fingerprints. If **P4TRUST** is unset, this file is **.p4trust** in the user's home directory. For more information, see "SSL-encrypted connections" on page 19.

If your installation requires plaintext (in order to support older Perforce applications), set **P4PORT** to **tcp:*hostname*:*port***.

Connecting over IPv6 networks

As of Release 2013.1, Perforce supports connectivity over IPv6 networks as well as over IPv4 networks.

Depending on the configuration of your LAN or WAN, your system administrator may recommend different port settings. Your administrator may also recommend that you set the `net.rfc3484` configurable to 1, either from the command line or in a `P4CONFIG` file:

```
p4 configure set net.rfc3484=1
```

Doing so ensures RFC3484-compliant behavior if the protocol value is not explicitly specified; that is, if the client-side configurable `net.rfc3484` is set to `1`, and `P4PORT` is set to `example.com:1666`, or `tcp:example.com:1666`, or `ssl:example.com:1666`, the user's operating system automatically determines, for any given connection, whether to use IPv4 or IPv6 when communicating with the versioning service.

Further information is available in the *Perforce Server Administrator's Guide: Fundamentals*.

Refining workspace views

By default, when you create a client workspace, the entire depot is mapped to your workspace. You can refine this mapping to view only a portion of the depot and to change the correspondence between depot and workspace locations.

To display or modify a workspace view, issue the `p4 client` command. Perforce displays the client workspace specification form, which lists mappings in the `View:` field:

```
Client:  bruno_ws
Owner:   bruno
Description:
         Created by bruno.
Root:    C:\bruno_ws
Options: noallwrite noclobber nocompress unlocked nomodtime normdir
SubmitOptions:   submitunchanged
View:
         //depot/...   //bruno_ws/...
```

The following sections provide details about specifying the client workspace view. For more information, see the `p4 client` command description and the description of views in the *P4 Command Reference*.

Specifying mappings

Views consist of multiple *mappings*. Each mapping has two parts.

- The left-hand side specifies one or more files in the depot and has the form: *`//depotname/file_specification`*
- The right-hand side specifies one or more files in the client workspace and has the form:*`//clientname/file_specification`*

The left-hand side of a client workspace view mapping is called the *depot side*, and the right-hand side is the *client side*.

To determine the location of any workspace file on your workstation, substitute the client workspace root for the workspace name on the client side of the mapping. For example, if the workspace root is `C:\bruno_ws`, the file `//depot/dev/main/jam/Jamfile` resides in `C:\bruno_ws\dev\main\jam\Jamfile`.

Later mappings override earlier ones. In the following example, the second line overrides the first line, mapping the files in `//depot/dev/main/docs/manuals/` up two levels. When files in `//depot/dev/main/docs/manuals/` are synced, they reside in `c:\bruno_ws\docs\`.

```
View:
    //depot/dev/...            //bruno_ws/dev/...
    //depot/dev/main/docs/...  //bruno_ws/docs/...
```

Using wildcards in workspace views

To map groups of files in workspace views, you use Perforce wildcards. Any wildcard used on the depot side of a mapping must be matched with an identical wildcard in the mapping's client side. You can use the following wildcards to specify mappings in your client workspace.

Wildcard	Description
`*`	Matches anything except slashes. Matches only within a single directory. Case sensitivity depends on your platform.
`...`	Matches anything including slashes. Matches recursively (everything in and below the specified directory).
`%%1 - %%9`	Positional specifiers for substring rearrangement in filenames.

In this simple client workspace view:

`//depot/dev/... //bruno_ws/dev/...`

all files in the depot's `dev` branch are mapped to the corresponding locations in the client workspace. For example, the file `//depot/dev/main/jam/Makefile` is mapped to the workspace file `C:\bruno_ws\dev\main\jam\Makefile`.

Note	To avoid mapping unwanted files, always precede the "`...`" wildcard with a forward slash.

The mappings in workspace views always refer to the locations of files and directories in the depot; you cannot refer to specific revisions of a file in a workspace view.

Mapping part of the depot

If you are interested only in a subset of the depot files, map that portion. Reducing the scope of the workspace view also ensures that your commands do not inadvertently affect the entire depot. To restrict the workspace view, change the left-hand side of the `View:` field to specify the relevant portion of the depot.

Example 2.3. Mapping part of the depot to the client workspace.

Dai is working on the Jam project and maintaining the web site, so she sets the `View:` field as follows:

```
View:
    //depot/dev/main/jam/...  //dai-beos-locust/jam/...
    //depot/www/live/...      //dai-beos-locust/www/live/...
```

Mapping files to different locations in the workspace

Views can consist of multiple mappings, which are used to map portions of the depot file tree to different parts of the workspace file tree. If there is a conflict in the mappings, later mappings have precedence over the earlier ones.

Example 2.4. Multiple mappings in a single workspace view.

The following view ensures that Microsoft Word files in the manuals folder reside in the workspace in a top-level folder called `wordfiles`.

```
View:
    //depot/...                          //bruno_ws/...
    //depot/dev/main/docs/manuals/*.doc  //bruno_ws/wordfiles/*.doc
```

Mapping files to different filenames

Mappings can be used to make the filenames in the workspace differ from those in the depot.

Example 2.5. Files with different names in the depot and the workspace

The following view maps the depot file `RELNOTES` to the workspace file `rnotes.txt`:

```
View:
    //depot/...                   //bruno_ws/...
    //depot/dev/main/jam/RELNOTES //bruno_ws/dev/main/jam/rnotes.txt
```

Rearranging parts of filenames

Positional specifiers `%%0` through `%%9` can be used to rearrange portions of filenames and directories.

Example 2.6. Using positional specifiers to rearrange filenames and directories.

The following view maps the depot file `//depot/allfiles/readme.txt` to the workspace file `filesbytype/txt/readme`:

```
View:
    //depot/allfiles/%%1.%%2  //bruno_ws/filesbytype/%%2/%%1
```

Excluding files and directories

Exclusionary mappings enable you to explicitly exclude files and directories from a workspace. To exclude a file or directory, precede the mapping with a minus sign (`-`). White space is not allowed between the minus sign and the mapping.

Example 2.7. Using views to exclude files from a client workspace.

Earl, who is working on the Jam project, does not want any HTML files synced to his workspace. His workspace view looks like this:

```
View:
    //depot/dev/main/jam/...        //earl-dev-beech/jam/...
    -//depot/dev/main/jam/....html  //earl-dev-beech/jam/....html
```

Restricting access by changelist

You can restrict access to depot paths to a particular point in time by providing the depot path names and changelist numbers in the `ChangeView` field of the client workspace specification. Files specified for the `ChangeView` field are read-only: they can be opened but not submitted. For example:

```
ChangeView:
      //depot/path/...@1000
```

In this example, revisions of the files in `//depot/path/...` are not visible if they were submitted after changelist 1000. Files submitted up to and including changelist 1000 are visible but read-only. You can specify multiple paths.

Avoiding mapping conflicts

When you use multiple mappings in a single view, a single file can inadvertently be mapped to two different places in the depot or workspace. When two mappings conflict in this way, the later mapping overrides the earlier mapping.

Example 2.8. Erroneous mappings that conflict.

Joe has constructed a view as follows:

```
View:
    //depot/proj1/...    //joe/project/...
    //depot/proj2/...    //joe/project/...
```

The second mapping `//depot/proj2/...` maps to `//joe/project` and conflicts with the first mapping. Because these mappings conflict, the first mapping is ignored; no files in `//depot/proj1` are mapped into the workspace: `//depot/proj1/file.c` is not mapped, even if `//depot/proj2/file.c` does not exist.

Mapping different depot locations to the same workspace location

Overlay mappings enable you to map files from more than one depot directory to the same place in a workspace. To overlay the contents of a second directory in your workspace, use a plus sign (**+**) in front of the mapping.

Example 2.9. Overlaying multiple directories in the same workspace.

Joe wants to combine the files from his projects when they are synced to his workspace, so he has constructed a view as follows:

```
View:
    //depot/proj1/...    //joe/project/...
    +//depot/proj2/...   //joe/project/...
```

The overlay mapping `+//depot/proj2/...` maps to `//joe/project`, and overlays the first mapping. Overlay mappings do not conflict. Files (even deleted files) in `//depot/proj2` take precedence over files in `//depot/proj1`. If `//depot/proj2/file.c` is missing (as opposed to being present, but deleted), then `//depot/proj1/file.c` is mapped into the workspace instead.

Overlay mappings are useful for applying sparse patches in build environments.

Dealing with spaces in filenames and directories

Use quotation marks to enclose files or directories that contain spaces.

Example 2.10. Dealing with spaces in filenames and directories.

Joe wants to map files in the depot into his workspace, but some of the paths contain spaces:

```
View:
    "//depot/Release 2.0/..."   //joe/current/...
    "//depot/Release 1.1/..."   "//joe/Patch Release/..."
    //depot/webstats/2011/...   "//joe/2011 Web Stats/..."
```

By placing quotation marks around the path components on the server side, client side, or both sides of the mappings, Joe can specify file names and/or directory components that contain spaces.

For more information, see "Spaces in filenames, pathnames, and identifiers" on page 28.

Mapping Windows workspaces across multiple drives

To specify a workspace that spans multiple Windows drives, use a `Root:` of `null` and specify the drive letters (in lowercase) in the workspace view. For example:

```
Client:     bruno_ws
Update:     2011/11/29 09:46:53
Access:     2011/03/02 10:28:40
Owner:      bruno
Root:       null
Options:    noallwrite noclobber nocompress unlocked nomodtime normdir
SubmitOptions: submitunchanged
LineEnd:    local
View:
    //depot/dev/...      "//bruno_ws/c:/Current Release/..."
    //depot/release/...  "//bruno_ws/d:/Prior Releases/..."
    //depot/www/...      //bruno_ws/d:/website/...
```

Using the same workspace from different machines

By default, you can only use a workspace on the machine that is specified by the **Host:** field. If you want to use the same workspace on multiple machines with different platforms, delete the **Host:** entry and set the **AltRoots:** field in the client workspace specification. You can specify a maximum of two alternate workspace roots. The locations must be visible from all machines that will be using them, for example through NFS or Samba mounts.

Perforce compares the current working directory against the main **Root:** first, and then against the two **AltRoots:** if specified. The first root to match the current working directory is used. If no roots match, the main root is used.

Note	If you are using a Windows directory in any of your workspace roots, specify the Windows directory as your main client **Root:** and specify your other workspace root directories in the **AltRoots:** field.

In the following example, if user **bruno**'s current working directory is located under **/usr/bruno**, Perforce uses the UNIX path as his workspace root, rather than **c:\bruno_ws**. This approach allows **bruno** to use the same client workspace specification for both UNIX and Windows development.

```
Client: bruno_ws
Owner:  bruno
Description:
        Created by bruno.
Root:   c:\bruno_ws
AltRoots:
        /usr/bruno/
```

To find out which client workspace root is in effect, issue the **p4 info** command and check the **Client root:** field.

If you edit text files in the same workspace from different platforms, ensure that the editors and settings you use preserve the line endings. For details about line-endings in cross-platform settings, see "Configuring line-ending settings" on page 18.

Automatically pruning empty directories from a workspace

By default, Perforce does not remove empty directories from your workspace. To change this behavior, issue the `p4 client` command and in the `Options:` field, change the option `normdir` to `rmdir`.

For more about changing workspace options, see "Configuring workspace options" on page 16.

Changing the location of your workspace

To change the location of files in your workspace, issue the `p4 client` command and change either or both of the `Root:` and `View:` fields. Before changing these settings, ensure that you have no files checked out (by submitting or reverting open files).

If you intend to modify both fields, perform the following steps to ensure that your workspace files are located correctly:

1. To remove the files from their old location in the workspace, issue the `p4 sync ...#none` command.
2. Change the `Root:` field. (The new client workspace root directory must exist on your workstation before you can retrieve files into it.)
3. To copy the files to their new locations in the workspace, perform a `p4 sync`. (If you forget to perform the `p4 sync ...#none` before you change the workspace view, you can always remove the files from their client workspace locations manually).
4. Change the `View:` field.
5. Again, perform a `p4 sync`. The files in the client workspace are synced to their new locations.

Configuring workspace options

The following table describes workspace `Options:` in detail.

Option	Description	Default
`[no]allwrite`	Specifies whether unopened files are always writable. By default, Perforce makes unopened files read-only. To avoid inadvertently overwriting changes or causing syncs to fail, specify `noallwrite`. A setting of `allwrite` leaves unopened files writable by the current user; it does not set filesystem permissions to ensure writability by any user of a multiuser system. If `allwrite` and `noclobber` are both set, Perforce performs a safe sync, comparing the content in your client workspace against what was last synced. If the file was modified outside of Perforce control, an error message is displayed and the file is not overwritten.	`noallwrite`

Option	Description	Default
`[no]clobber`	Specifies whether `p4 sync` overwrites writable but unopened workspace files. (By default, Perforce does not overwrite unopened files if they are writable.) If `allwrite` and `noclobber` are both set, Perforce performs a safe sync, comparing the content in your client workspace against what was last synced. If the file was modified outside of Perforce control, an error message is displayed and the file is not overwritten.	`noclobber`
`[no]compress`	Specifies whether data is compressed when it is sent between your workstation and the Perforce service.	`nocompress`
`[un]locked`	Specifies whether other users can use, edit, or delete the client workspace specification. A Perforce administrator can override the lock with the `-f` (force) option. If you lock your client workspace specification, be sure to set a password for the workspace's owner using the `p4 passwd` command.	`unlocked`
`[no]modtime`	For files *without* the `+m` (modtime) file type modifier, if `modtime` is set, the modification date (on the local filesystem) of a newly synced file is the datestamp *on the file* when the file was submitted to the depot. If `nomodtime` is set, the modification date is the date and time of sync. For files *with* the `+m` (modtime) file type, the modification date (on the local filesystem) of a newly synced file is the datestamp on the file when the file was submitted to the depot, regardless of the setting of `modtime` or `nomodtime` on the client.	`nomodtime` (date and time of sync). Ignored for files with the `+m` file type modifier.
`[no]rmdir`	Specifies whether `p4 sync` deletes empty directories in a workspace if all files in the directory have been removed.	`normdir`

Configuring submit options

To control what happens to files in a changelist when you submit the changelist to the depot, set the `SubmitOptions:` field. Valid settings are as follows.

Option	Description
`submitunchanged`	All open files (with or without changes) are submitted to the depot. This is the default behavior of Perforce.
`submitunchanged+reopen`	All open files (with or without changes) are submitted to the depot, and all files are automatically reopened in the default changelist.

Option	Description
`revertunchanged`	Only those files with content, type, or resolved changes are submitted to the depot. Unchanged files are reverted.
`revertunchanged+reopen`	Only those files with content, type, or resolved changes are submitted to the depot and reopened in the default changelist. Unchanged files are reverted and *not* reopened in the default changelist.
`leaveunchanged`	Only those files with content, type, or resolved changes are submitted to the depot. Any unchanged files are moved to the default changelist.
`leaveunchanged+reopen`	Only those files with content, type, or resolved changes are submitted to the depot. Unchanged files are moved to the default changelist, and changed files are reopened in the default changelist. This option is similar to **`submitunchanged+reopen`**, except that no unchanged files are submitted to the depot.

Configuring line-ending settings

To specify how line endings are handled when you sync text files, set the **`LineEnd:`** field. Valid settings are as follows:

Option	Description
`local`	Use mode native to the client (default)
`unix`	UNIX-style (and Mac OS X) line endings: **`LF`**
`mac`	Mac pre-OS X: **`CR`** only
`win`	Windows- style: **`CR`**, **`LF`**
`share`	The **`share`** option normalizes mixed line-endings into UNIX line-end format. The **`share`** option does not affect files that are synced into a client workspace; however, when files are submitted back to the Perforce service, the share option converts all Windows-style **`CR/LF`** line-endings and all Mac-style **`CR`** line-endings to the UNIX-style **`LF`**, leaving lone **`LF`**s untouched. When you sync your client workspace, line endings are set to **`LF`**. If you edit the file on a Windows machine, and your editor inserts **`CR`**'s before each **`LF`**, the extra **`CR`**'s do not appear in the archive file. The most common use of the **`share`** option is for users of Windows workstations who mount their UNIX home directories as network drives; if you sync files from UNIX, but edit the files on a Windows machine.

For detailed information about how Perforce uses the line-ending settings, see "CR/LF Issues and Text Line-endings" in the Perforce knowledge base:

http://answers.perforce.com/articles/KB_Article/CR-LF-Issues-and-Text-Line-endings

Deleting client workspace specifications

To delete a workspace, issue the `p4 client -d clientname` command. Deleting a client workspace removes Perforce's record of the workspace but does not remove files from the workspace or the depot.

When you delete a workspace specification:

1. Revert (or submit) any pending or shelved changelists associated with the workspace.
2. Delete existing files from a client workspace (`p4 sync ...#none`). (optional)
3. Delete the workspace specification.

If you delete the workspace specification before you delete files in the workspace, you can delete workspace files using your operating system's file deletion command.

Security

For security purposes, your Perforce administrator can configure the Perforce service to require SSL-encrypted connections, user passwords, and to limit the length of time for which your login ticket is valid. The following sections provide details.

SSL-encrypted connections

If your installation requires SSL, make sure your `P4PORT` is of the form `ssl:hostname:port`. If you attempt to communicate in plaintext with an SSL-enabled Perforce server, the following error message is displayed:

```
Failed client connect, server using SSL.
Client must add SSL protocol prefix to P4PORT.
```

Set `P4PORT` to `ssl:hostname:port`, and attempt to reconnect to the server.

The first time you establish an encrypted connection with an SSL-enabled server, you are prompted to verify the server's fingerprint:

```
The authenticity of '10.0.0.2:1818' can't be established,
this may be your first attempt to connect to this P4PORT.
The fingerprint for the key sent to your client is
CA:BE:5B:77:14:1B:2E:97:F0:5F:31:6E:33:6F:0E:1A:E9:DA:EF:E2
```

Your administrator can confirm whether the displayed fingerprint is correct or not. If (and only if) the fingerprint is correct, use the `p4 trust` command to add it to your `P4TRUST` file. If `P4TRUST` is unset, this file is assumed to be `.p4trust` in your home directory:

```
$ p4 trust
The fingerprint of the server of your P4PORT setting
'ssl:example.com:1818' (10.0.0.2:1818) is not known.
That fingerprint is
CA:BE:5B:77:14:1B:2E:97:F0:5F:31:6E:33:6F:0E:1A:E9:DA:EF:E2
Are you sure you want to establish trust (yes/no)?
Added trust for P4PORT 'ssl:example.com:1818' (10.0.0.2:1818)
```

If the fingerprint is accurate, enter `yes` to trust this server. You can also install a fingerprint directly into your trust file from the command line. Run:

`p4 trust -p ssl:`*`hostname`*`:`*`port`*` -i `*`fingerprint`*

where `ssl:`*`hostname`*`:`*`port`* corresponds to your `P4PORT` setting, and ***`fingerprint`*** corresponds to a fingerprint that your administrator has verified.

From this point forward, any SSL connection to `ssl:example.com:1818` is trusted, so long as the server at `example.com:1818` continues to report a fingerprint that matches the one recorded in your `P4TRUST` file.

If the Perforce server ever reports a different fingerprint than the one that you have trusted, the following error message is displayed:

```
******* WARNING P4PORT IDENTIFICATION HAS CHANGED! *******
It is possible that someone is intercepting your connection
to the Perforce P4PORT '10.0.50.39:1667'
If this is not a scheduled key change, then you should contact
your Perforce administrator.
The fingerprint for the mismatched key sent to your client is
18:FC:4F:C3:2E:FA:7A:AE:BC:74:58:2F:FC:F5:87:7C:BE:C0:2D:B5
To allow connection use the 'p4 trust' command.
```

This error message indicates that the server's fingerprint has changed from one that you stored in your `P4TRUST` file and indicates that the server's SSL credentials have changed.

Although the change to the fingerprint may be legitimate (for example, your administrator controls the length of time for which your server's SSL credentials remain valid, and your server's credentials may have expired), it can also indicate the presence of a security risk.

Warning

If you see this error message, and your Perforce administrator has not notified you of a change to your server's key and certificate pair, it is imperative that you *independently* verify the accuracy of the reported fingerprint.

Unless you can independently confirm the veracity of the new fingerprint (by some out-of-band means ranging from the company's intranet site, or by personally contacting your administrator), do not trust the changed fingerprint.

Connecting to services that require plaintext connections

If your Perforce installation requires plaintext (in order to support older Perforce applications), set `P4PORT` to `tcp:hostname:port`. If you attempt to use SSL to connect to a service that expects plaintext connections, the following error message is displayed:

```
Perforce client error:
    SSL connect to ssl:host:port failed (Connection reset by peer).
    Remove SSL protocol prefix from P4PORT.
```

Set `P4PORT` to `tcp:hostname:port` (or, if you are using applications at release 2011.1 or earlier, set `P4PORT` to `hostname:port`), and attempt to reconnect to the service.

Passwords

Depending on the security level at which your Perforce installation is running, you might need to log in to Perforce before you can run Perforce commands. Without passwords, any user can assume the identity of any other Perforce user by setting `P4USER` to a different user name or specifying the `-u` option when you issue a `p4` command. To improve security, use passwords.

Setting passwords

To create a password for your Perforce user, issue the `p4 passwd` command.

Passwords may be up to 1024 characters in length. Your system administrator can configure Perforce to require "strong" passwords, the minimum length of a password, and if you have been assigned a default password, your administrator can further require that you change your password before you first use Perforce.

By default, the Perforce service defines a password as strong if it is at least eight characters long and contains at least two of the following:

- Uppercase letters
- Lowercase letters
- Non-alphabetic characters

In an environment with a minimum password length of eight characters, for example, `a1b2c3d4`, `A1B2C3D4`, `aBcDeFgH` would be considered strong passwords.

To reset or remove a password (without knowing the password), Perforce superuser privilege is required. If you need to have your password reset, contact your Perforce administrator. See the *Perforce Server Administrator's Guide: Fundamentals* for details.

Using your password

If your Perforce user has a password set, you must use it when you issue `p4` commands. To use the password, you can:

- Log into Perforce by issuing the `p4 login` command, before issuing other commands.
- Set `P4PASSWD` to your password, either in the environment or in a config file.
- Specify the `-P` *`password`* option when you issue `p4` commands (for instance, `p4 -P` *`mypassword`* `submit`).
- Windows or OS X: store your password by using the `p4 set -s` command. Not advised for sites where security is high. Perforce administrators can disable this feature.

Connection time limits

Your Perforce administrator can configure the Perforce service to enforce time limits for users. Perforce uses ticket-based authentication to enforce time limits. Because ticket-based authentication does not rely on environment variables or command-line options, it is more secure than password-based authentication.

Tickets are stored in a file in your home directory. After you have logged in, your ticket is valid for a limited period of time (by default, 12 hours).

Logging in and logging out

If time limits are in effect at your site, you must issue the `p4 login` command to obtain a ticket. Enter your password when prompted. If you log in successfully, a ticket is created for you in the ticket file in your home directory, and you are not prompted to log in again until either your ticket expires or you log out by issuing the `p4 logout` command.

To see how much time remains before your login expires, issue the following command:

```
p4 login -s
```

If your ticket is valid, the length of time remaining is displayed. To extend a ticket's lifespan, use `p4 login` while already logged in. Your ticket's lifespan is extended by 1/3 of its initial timeout setting, subject to a maximum of your ticket's initial timeout setting.

To log out of Perforce, issue the following command:

```
p4 logout
```

Working on multiple machines

By default, your ticket is valid only for the IP address of the machine from which you logged in. If you use Perforce from multiple machines that share a home directory (typical in many UNIX environments), log in with:

```
p4 login -a
```

Using `p4 login -a` creates a ticket in your home directory that is valid from all IP addresses, enabling you to remain logged into Perforce from more than one machine.

To log out from all machines simultaneously, issue the following command:

```
p4 logout -a
```

For more information about the `p4 login` and `p4 logout` commands, see the *P4 Command Reference*.

Working with Unicode

The Perforce service can be run in Unicode mode to activate support for file names or directory names that contain Unicode characters, and Perforce identifiers (for example, user names) and specifications (for example, changelist descriptions or jobs) that contain Unicode characters.

In Unicode mode, the Perforce service also translates `unicode` files and metadata to the character set configured on the user's workstation, and verifies that the unicode files and metadata contain valid UTF-8 characters.

> **Note**
>
> If you only need to manage textual files that contain Unicode characters, but do not need the features listed above, you do not need to run Perforce in Unicode mode. Your system administrator will tell you if your site is using Unicode mode or not.
>
> For these installations, assign the Perforce `utf16` file type to textual files that contain Unicode characters. You do not have to set the `P4CHARSET` or `P4COMMANDCHARSET` environment variables. See "Assigning File Types for Unicode Files" on page 114 for details.

To correctly interoperate in Unicode mode, and to ensure that such files are translated correctly by the Perforce service when the files are synced or submitted, you must set `P4CHARSET` to the character set that corresponds to the format used on your workstation by the applications that access them, such as text editors or IDEs. These formats are typically listed when you save the file using the **Save As...** menu option.

Values of `P4CHARSET` that begin with `utf16` or `utf32` further require that you also set `P4COMMANDCHARSET` to a non `utf16` or `utf32` character set in which you want server output displayed. "Server output" includes informational and error messages, diff output, and information returned by reporting commands.

For a complete list of valid `P4CHARSET` values, issue the command `p4 help charset`.

For further information, see the *Perforce Server Administrator's Guide: Fundamentals*.

Setting P4CHARSET on Windows

To set `P4CHARSET` for all users on a workstation, you need Windows administrator privileges. Issue the following command:

```
p4 set -s P4CHARSET=character_set
```

To set `P4CHARSET` for the user currently logged in:

```
p4 set P4CHARSET=character_set
```

Your workstation must have a compatible TrueType or OpenType font installed.

Setting P4CHARSET on UNIX

You can set P4CHARSET from a command shell or in a startup script such as .kshrc, .cshrc, or .profile. To determine the proper value for P4CHARSET, examine the setting of the LANG or LOCALE environment variable. Common settings are as follows

If LANG is...	Set P4CHARSET to
en_US.ISO_8859-1	iso8859-1
ja_JP.EUC	eucjp
ja_JP.PCK	shiftjis

In general, for a Japanese installation, set P4CHARSET to eucjp, and for a European installation, set P4CHARSET to iso8859-1.

Chapter 3 Issuing P4 Commands

This chapter provides basic information about `p4` commands, including command-line syntax, arguments, and options. For full details about command syntax, refer to the *P4 Command Reference*.

Certain commands require administrator or superuser permission. For details, consult the *Perforce Server Administrator's Guide: Fundamentals*

Command-line syntax

The basic syntax for commands is as follows:

`p4 [global options] command [command-specific options] [command arguments]`

The following options can be used with all `p4` commands.

Global options	Description and Example
`-c` *clientname*	Specifies the client workspace associated with the command. Overrides `P4CLIENT`. `p4 -c bruno_ws edit //depot/dev/main/jam/Jambase`
`-C` *charset*	Specifies the client workspace's character set. Overrides `P4CHARSET`. `p4 -C utf8 sync`
`-d` *directory*	Specifies the current directory, overriding the environment variable `PWD`. `p4 -d ~c:\bruno_ws\dev\main\jam\Jambase Jamfile`
`-G`	Format all output as marshaled Python dictionary objects (for scripting with Python). `p4 -G info`
`-H` *host*	Specifies the hostname of the client workstation, overriding `P4HOST`. `p4 -H deneb print //depot/dev/main/jam/Jambase`
`-I`	Specify that progress indicators, if available, are desired. This option is not compatible with the `-s` and `-G` options. At present, the progress indicator is only supported by two commands: submitting a changelist with `p4 -I submit` and "quietly" syncing files with `p4 -I sync -q`.
`-L` *language*	Specifies the language to use for error messages from the Perforce service. Overrides `P4LANGUAGE`. In order for this option to work, your administrator must have loaded support for non-English messages in the database. `p4 -L` *language* `info`

Global options	Description and Example
`-p` *port*	Specifies the protocol, host and port number used to connect to the Perforce service, overriding `P4PORT`. `p4 -p ssl:deneb:1818 clients`
`-P` *password*	Supplies a Perforce password, overriding `P4PASSWD`. Usually used in combination with the `-u` *username* option. `p4 -u earl -P secretpassword job`
`-r` *retries*	Specifies the number of times to retry a command (notably, `p4 sync`) if the network times out.
`-Q` *charset*	Specifies the character set to use for command input and output; if you have set `P4CHARSET` to a UTF-16 or UTF-32 value, you must set `P4COMMANDCHARSET` to a non-UTF-16 or -32 value in order to use the `p4` command-line client. `p4 -Q utf32 -C utf8 sync`
`-s`	Prepend a tag to each line of output (for scripting purposes). `p4 -s info`
`-u` *username*	Specifies a Perforce user, overriding `P4USER`. `p4 -u bill user`
`-x` *filename*	Read arguments, one per line, from the specified file. To read arguments from standard input, specify `-x -`. `p4 -x myargs.txt`
`-z tag`	To facilitate scripting, displays the output of reporting commands in the format as that generated by `p4 fstat`. `p4 -z tag info`
`-q`	Quiet mode; suppress all informational message and report only warnings or errors.
`-V`	Displays the version of the `p4` executable.

To display the options for a specific command, issue the `p4 help` command. For example:

```
$ p4 help add

    add -- Open a new file to add it to the depot

    p4 add [ -c changelist# ] [ -d -f -I -n ] [ -t filetype ] file ...

     Open a file for adding to the depot.  If the file exists on the
     client, it is read to determine if it is text or binary. If it does
     not exist, it is assumed to be text.  To be added, the file must not
     already reside in the depot, or it must be deleted at the current
     head revision.  Files can be deleted and re-added.
[…]
```

For the full list of global options, commands, and command-specific options, see the *P4 Command Reference*.

Specifying filenames on the command line

Much of your everyday use of Perforce consists of managing files. You can specify filenames in `p4` commands as follows:

- **Local syntax**: the file's name as specified in your local shell or operating system.

  Filenames can be specified using an absolute path (for example, `c:\bruno_ws\dev\main\jam\fileos2.c`) or a path that is relative to the current directory (for example, `.\jam\fileos2.c`).

  Relative components (`.` or `..`) cannot be specified following fixed components. For example, `mysub/mydir/./here/file.c` is invalid, because the dot (`.`) follows the fixed `mysub/mydir` components.

- **Depot syntax**: use the following format: `//depotname/file_path`, specifying the pathname of the file relative to the depot root directory. Separate the components of the path using forward slashes. For example: `//depot/dev/main/jam/Jambase`.

- **Client syntax**: use the following format: `//workspacename/file_path`, specifying the pathname of the file relative to the client root directory. Separate the components of the path using forward slashes. For example: `//ona-agave/dev/main/jam/Jambase`.

Example 3.1. Using different syntaxes to refer to the same file

Local syntax: `p4 delete c:\bruno_ws\dev\main\jam\Jambase`

Depot syntax: `p4 delete //depot/dev/main/jam/Jambase`

Client syntax: `p4 delete //bruno_ws/dev/main/jam/Jambase`

Perforce wildcards

For commands that operate on sets of files, Perforce supports two wildcards.

Wildcard	Description
*	Matches anything except slashes. Matches only within a single directory. Case sensitivity depends on your platform.
...	Matches anything including slashes. Matches recursively (everything in and below the specified directory).

Perforce wildcards can be used with local or Perforce syntax, as in the following examples.

Expression	Matches
`J*`	Files in the current directory starting with `J`.
`*/help`	All files called `help` in current subdirectories.
`./...`	All files under the current directory and its subdirectories.
`./....c`	All files under the current directory and its subdirectories, that end in `.c`.
`/usr/bruno/...`	All files under `/usr/bruno`.
`//bruno_ws/...`	All files in the workspace or depot that is named `bruno_ws`.
`//depot/...`	All files in the depot named `depot`.
`//...`	All files in all depots.

The * wildcard is expanded locally by the operating system before the command is sent to the Perforce service. To prevent the local operating system from expanding the * wildcard, enclose it in quotes or precede it with a backslash.

Note

The "`...`" wildcard cannot be used with the `p4 add` command. The "`...`" wildcard is expanded by the Perforce service, and, because the service cannot determine which files are being added, it can't expand the wildcard. The * wildcard can be used with `p4 add`, because it is expanded by the operating system shell and not by Perforce.

Restrictions on filenames and identifiers

Spaces in filenames, pathnames, and identifiers

Use quotation marks to enclose files or directories that contain spaces. For example:

```
"//depot/dev/main/docs/manuals/recommended configuration.doc"
```

If you specify spaces in names for other Perforce objects, such as branch names, client names, label names, and so on, the spaces are automatically converted to underscores by the Perforce service.

Length limitations

Names assigned to Perforce objects such as branches, client workspaces, and so on, cannot exceed 1,024 characters.

Reserved characters

By default, the following reserved characters are not allowed in Perforce identifiers or names of files managed by Perforce.

Reserved Character	Reason
`@`	File revision specifier for date, label name, or changelist number
`#`	File revision numbers
`*`	Wildcard
`...`	Wildcard (recursive)
`%%1 - %%9`	Wildcard (positional)
`/`	Separator for pathname components

These characters have conflicting and secondary uses. Conflicts include the following:

- UNIX separates path components with `/`, but many DOS commands interpret `/` as a command-line switch.
- Most UNIX shells interpret `#` as the beginning of a comment.
- Both DOS and UNIX shells automatically expand `*` to match multiple files, and the DOS command line uses `%` to refer to variables.

To specify these characters in filenames or paths, use the ASCII expression of the character's hexadecimal value, as shown in the following table.

Character	ASCII
`@`	`%40`
`#`	`%23`
`*`	`%2A`
`%`	`%25`

Specify the filename literally when you add it; then use the ASCII expansion to refer to it thereafter. For example, to add a file called `recommended@configuration.doc`, issue the following command:

```
p4 add -f //depot/dev/main/docs/manuals/recommended@configuration.doc
```

When you submit the changelist, the characters are automatically expanded and appear in the change submission form as follows:

```
//depot/dev/main/docs/manuals/recommended%40configuration.doc
```

After you submit the changelist with the file's addition, you must use the ASCII expansion to sync the file to your workspace or to edit it within your workspace. For example:

```
p4 sync //depot/dev/main/docs/manuals/recommended%40configuration.doc
```

The requirement to escape the special characters `@`, `#`, `*`, or `%` also applies if you attempt to use them in the `Root:` or `AltRoots:` fields of your client workspace specification; escape them with `%40`, `%23`, `%2A`, or `%25` respectively.

Filenames containing extended (non-ASCII) characters

Non-ASCII characters are allowed in filenames and Perforce identifiers, but entering them from the command line might require platform-specific solutions. If you are using Perforce in unicode mode, all users must have `P4CHARSET` set properly. For details about setting `P4CHARSET`, see the *P4 Command Reference* and the *Internationalization Notes*.

In international environments, use a common code page or locale setting to ensure that all filenames are displayed consistently across all machines in your organization. To set the code page or locale:

- Windows: use the **Regional Settings** applet in the **Control Panel**
- UNIX: set the `LOCALE` environment variable

Specifying file revisions

Each time you submit a file to the depot, its revision number is incremented. To specify revisions prior to the most recent, use the `#` revision specifier to specify a revision number, or`@` to specify a date, changelist, client workspace, or label corresponding to the version of the file you are working on. Revision specifications can be used to limit the effect of a command to specified file revisions.

Warning: Some operating system shells treat the Perforce revision character `#` as a comment character if it starts a word. If your shell is one of these, escape the `#` when you use it in `p4` commands.

The following table describes the various ways you can specify file revisions.

Revision needed	Syntax and example
Revision number	*`file#n`* Example: `p4 sync //depot/dev/main/jam/Jambase#3` Refers to revision 3 of file `Jambase`
The revision submitted as of a specified changelist	*`file@changelist_number`* Examples:

Revision needed	Syntax and example

	`p4 sync //depot/dev/main/jam/Jambase@126`
	
	Refers to the version of `Jambase` when changelist 126 was submitted, even if it was not part of the change.
	
	`p4 sync //depot/...@126`
	
	Refers to the state of the entire depot at changelist 126 (numbered changelists are explained in "Managing changelists" on page 42).

The revision in a specified label	*`file`*`@`*`labelname`* Example: `p4 sync //depot/dev/main/jam/Jambase@beta` The revision of `Jambase` in the label called `beta`. For details about labels, refer to Chapter 7, "Labels" on page 79.
The revision last synced to a specified client workspace	*`file`*`@`*`clientname`* Example: `p4 sync //depot/dev/main/jam/Jambase@bruno_ws` The revision of `Jambase` last synced to client workspace `bruno_ws`
Remove the file	*`file`*`#none` Example: `p4 sync //depot/dev/main/jam/Jambase#none` Removes `Jambase` from the client workspace.
The most recent version of the file	*`file`*`#head` Example: `p4 sync //depot/dev/main/jam/Jambase#head` Same as `p4 sync //depot/dev/main/jam/Jambase` (If you omit the revision specifier, the head revision is synced.)
The revision last synced to your workspace	*`file`*`#have` Example: `p4 files //depot/dev/main/jam/Jambase#have`

Revision needed	Syntax and example
The head revision of the file in the depot on the specified date	*`file@date`* Example: `p4 sync //depot/dev/main/jam/Jambase@2011/05/18` The head revision of `Jambase` as of midnight May 18, 2011.

The head revision of the file in the depot on the specified date at the specified time	`file@"date[:time]"`
	
	Example:
	
	`p4 sync //depot/dev/main/jam/Jambase@"2011/05/18"`
	
	Specify dates in the format *`YYYY/MM/DD`*. Specify time in the format *`HH:MM:SS`* using the 24-hour clock. Time defaults to 00:00:00.
	
	Separate the date and the time by a single space or a colon. (If you use a space to separate the date and time, you must also enclose the entire date-time specification in double quotes.)

Example 3.2. Retrieving files using revision specifiers

Bruno wants to retrieve all revisions that existed at changelist number 30. He types:

```
p4 sync //depot/dev/main/jam/Jambase@30
```

Another user can sync their workspace so that it contains the same file revisions Bruno has synced by specifying Bruno's workspace, as follows:

```
p4 sync @bruno_ws
```

Example 3.3. Removing all files from the client workspace

```
p4 sync ...#none
```

The files are removed from the workspace but not from the depot.

Date and time specifications

Date and time specifications are obtained from the time zone of the machine that hosts the Perforce service. To display the date, time, offset from GMT, and time zone in effect, issue the `p4 info` command. The versioning service stores times as the number of seconds since 00:00:00 GMT Jan. 1, 1970), so if you move across time zones, the times stored in the service are correctly reported in the new time zone.

Revision ranges

Some commands can operate on a range of file revisions. To specify a revision range, specify the start and end revisions separated by a comma, for example, `#3,4`.

The commands that accept revision range specifications are:

- `p4 annotate`
- `p4 changes`
- `p4 dirs`
- `p4 filelog`
- `p4 files`
- `p4 fixes`
- `p4 grep`
- `p4 integrate`
- `p4 interchanges`
- `p4 jobs`
- `p4 labels`
- `p4 labelsync`
- `p4 list`
- `p4 merge`
- `p4 print`
- `p4 sizes`
- `p4 sync`
- `p4 tag`

For the preceding commands:

- If you specify a single revision, the command operates on revision #1 through the revision you specify (except for **`p4 sync`**, **`p4 print`**, and **`p4 files`**, which operate on the highest revision in the range).
- If you omit the revision range entirely, the command affects all file revisions.

Example 3.4. Listing changes using revision ranges

A release manager needs to see a quick list of all changes made to the jam project in July 2010. He types:

```
p4 changes //depot/dev/main/jam/...@2010/7/1,2010/8/1
```

The resulting list of changes looks like this:

```
Change 673 on 2010/07/31 by bruno@bruno_ws 'Final build for QA'
Change 633 on 2010/07/1 by bruno@bruno_ws 'First build w/bug fix'
Change 632 on 2010/07/1 by bruno@bruno_ws 'Started work'
```

Reporting commands

The following table lists some useful reporting commands.

To display	Use this command
A list of `p4` commands with a brief description	`p4 help commands`
Detailed help about a specific *command*	`p4 help` *command*
Command line options common to all Perforce commands	`p4 help usage`
Details about Perforce view syntax	`p4 help views`
All the arguments that can be specified for the `p4 help` command	`p4 help`
The Perforce settings configured for your environment	`p4 info`
The file revisions in the client workspace	`p4 have`
Preview the results of a `p4 sync` (to see which files would be transferred)	`p4 sync -n`
Preview the results of a `p4 delete` (to see which files would be marked for deletion)	`p4 delete -n` *files*

Using Perforce forms

Some Perforce commands, for example `p4 client` and `p4 submit`, use a text editor to display a form into which you enter the information that is required to complete the command (for example, a description of the changes you are submitting). After you change the form, save it, and exit the editor, Perforce parses the form and uses it to complete the command. (To configure the text editor that is used to display and edit Perforce forms, set `P4EDITOR`.)

When you enter information into a Perforce form, observe the following rules:

- Field names (for example, `View:`) must be flush left (not indented) and must end with a colon.
- Values (your entries) must be on the same line as the field name, or indented with tabs on the lines beneath the field name.

Some field names, such as the `Client:` field in the `p4 client` form, require a single value; other fields, such as `Description:`, take a block of text; and others, like `View:`, take a list of lines.

Certain values, like **`Client:`** in the client workspace form, cannot be changed. Other fields, like **`Description:`** in **`p4 submit`**, *must* be changed. If you don't change a field that needs to be changed, or vice versa, Perforce displays an error. For details about which fields can be modified, see the *P4 Command Reference* or use **`p4 help command`**.

Chapter 4

Managing Files and Changelists

This chapter tells you how to manage files and work in a team development environment, where multiple users who are working on the same files might need to reconcile their changes.

Managing files

To change files in the depot (file repository), you open the files in changelists and submit the changelists with a description of your changes. Perforce assigns numbers to changelists and maintains the revision history of your files. This approach enables you to group related changes and find out who changed a file and why and when it was changed. Here are the basic steps for working with files.

Task	Description
Syncing (retrieving files from the depot)	Issue the `p4 sync` command, specifying the files and directories you want to retrieve from the depot. You can only sync files that are mapped in your client view.
Adding files to the depot	1. Create the file in the workspace. 2. Open the file for add in a changelist (`p4 add`). 3. Submit the changelist (`p4 submit`).
Editing files and checking in changes	1. If necessary, sync the desired file revision to your workspace (`p4 sync`). 2. Open the file for edit in a changelist (`p4 edit`). 3. Make your changes. 4. Submit the changelist (`p4 submit`). To discard changes, issue the `p4 revert` command.
Deleting files from the depot	1. Open the file for delete in a changelist (`p4 delete`). The file is deleted from your workspace. 2. Submit the changelist (`p4 submit`). The file is deleted from the depot.
Discarding changes	Revert the files or the changelist in which the files are open. Reverting has the following effects on open files: Add: no effect - the file remains in your workspace. Edit: the revision you opened is resynced from the depot, overwriting any changes you made to the file in your workspace. Delete: the file is resynced to your workspace.

Files are added to, deleted from, or updated in the depot only when you successfully submit the pending changelist in which the files are open. A changelist can contain a mixture of files open for add, edit and delete.

For details about the syntax that you use to specify files on the command line, refer to "Specifying filenames on the command line" on page 27. The following sections provide more details about working with files.

Syncing (retrieving) files

To retrieve files from the depot into your client workspace, issue the **p4 sync** command. You cannot sync files that are not in your client view. For details about specifying client views, see "Refining workspace views" on page 10.

Example 4.1. Copying files from the depot to a client workspace.

The following command retrieves the most recent revisions of all files in the client view from the depot into the workspace. As files are synced, they are listed in the command output.

```
C:\bruno_ws>p4 sync
//depot/dev/main/bin/bin.linux24x86/readme.txt#1 - added as c:\bruno_ws\dev\main\bin
\bin.linux24x86\readme.txt
//depot/dev/main/bin/bin.ntx86/glut32.dll#1 - added as c:\bruno_ws\dev\main\bin
\bin.ntx86\glut32.dll
//depot/dev/main/bin/bin.ntx86/jamgraph.exe#2 - added as c:\bruno_ws\dev\main\bin
\bin.ntx86\jamgraph.exe
[...]
```

The **p4 sync** command adds, updates, or deletes files in the client workspace to bring the workspace contents into agreement with the depot. If a file exists within a particular subdirectory in the depot, but that directory does not exist in the client workspace, the directory is created in the client workspace when you sync the file. If a file has been deleted from the depot, **p4 sync** deletes it from the client workspace.

To sync revisions of files prior to the latest revision in the depot, use revision specifiers. For example, to sync the first revision of **Jamfile**, which has multiple revisions, issue the following command:

```
p4 sync //depot/dev/main/jam/Jamfile#1
```

For more details, refer to "Specifying file revisions" on page 30.

To sync groups of files or entire directories, use wildcards. For example, to sync everything in and below the **jam** folder, issue the following command:

```
p4 sync //depot/dev/main/jam/...
```

For more details, see "Perforce wildcards" on page 27.

The Perforce service tracks which revisions you have synced. For maximum efficiency, Perforce does not resync an already-synced file revision. To resync files you (perhaps inadvertently) deleted manually, specify the **-f** option when you issue the **p4 sync** command.

Adding files

To add files to the depot, create the files in your workspace, then issue the **p4 add** command. The **p4 add** command opens the files for **add** in the default pending changelist. The files are added when you successfully submit the default pending changelist. You can open multiple files for add using a single `p4 add` command by using wildcards. You cannot use the Perforce `...` wildcard to add files recursively.

For platform-specific details about adding files recursively (meaning files in subdirectories), see "Adding a Directory Tree" in the Perforce knowledge base:

http://answers.perforce.com/articles/KB_Article/Adding-a-Directory-Tree

Example 4.2. Adding files to a changelist.

Bruno has created a couple of text files that he needs to add to the depot. To add all the text files at once, he uses the "*" wildcard when he issues the `p4 add` command.

```
C:\bruno_ws\dev\main\docs\manuals>p4 add *.txt
//depot/dev/main/docs/manuals/installnotes.txt#1 - opened for add
//depot/dev/main/docs/manuals/requirements.txt#1 - opened for add
```

Now the files he wants to add to the depot are open in his default changelist. The files are stored in the depot when the changelist is submitted.

Example 4.3. Submitting a changelist to the depot.

Bruno is ready to add his files to the depot. He types `p4 submit` and sees the following form in a standard text editor:

```
Change: new
Client: bruno_ws
User:   bruno
Status: new
Description:
        <enter description here>
Type:   public
Files:
    //depot/dev/main/docs/manuals/installnotes.txt   # add
    //depot/dev/main/docs/manuals/requirements.txt   # add
```

Bruno changes the contents of the `Description:` field to describe his file updates. When he's done, he saves the form and exits the editor, and the new files are added to the depot.

You must enter a description in the `Description:` field. You can delete lines from the `Files:` field. Any files deleted from this list are moved to the next default changelist, and are listed the next time you submit the default changelist.

If you are adding a file to a directory that does not exist in the depot, the depot directory is created when you successfully submit the changelist.

You can restrict a changelist from public view by changing the `Type:` field from `public` to `restricted`. In general, if a changelist is restricted, only those users with `list` access to at least one of the files in the changelist are permitted to see the changelist description.

Ignoring groups of files when adding

Sometimes development processes result in the creation of extraneous content that should not be submitted to the depot. Compilers produce object files and executables during development, text editors and word processors produce backup files, and you may have your own personal conventions for notes on work in progress.

To ignore files (or groups of files) when adding, create a file with a list of file specifications you wish to ignore, and set the `P4IGNORE` environment variable to point to this file.

When you add files, the full local path and parent directories of any file to be added are searched for `P4IGNORE` files. If any `P4IGNORE` files exist, their rules are added to a list, with greater precedence given to `P4IGNORE` rules closest to the file being added.

The syntax for `P4IGNORE` files is *not* the same as Perforce syntax. Instead, it is similar to that used by other versioning systems: files are specified in local syntax, a `#` character at the beginning of a line denotes a comment, a `!` character at the beginning of a line excludes the file specification, and the `*` wildcard matches substrings. The Perforce wildcard of "`...`" is not permitted.

Character	Meaning in P4IGNORE files
*	Matches anything except slashes. Matches only within a single directory. Case sensitivity depends on your client platform.
!	Exclude the file specification from consideration.
#	Comment character; this line is ignored.

Example 4.4. Ignoring groups of files when adding.

Bruno unit tests his code before submitting it to the depot and does not want to accidentally add any object files or generated executables when reconciling his workspace.

Bruno first sets `P4IGNORE` to point to the correct file:

```
export P4IGNORE=.p4ignore
```

He then creates the following file and stores it as `.p4ignore` in the root of his workspace:

```
# Ignore .p4ignore files
.p4ignore
# Ignore object files, shared libraries, executables
*.dll
*.so
*.exe
*.o
# Ignore all text files except readme file
*.txt
!readme.txt
```

The next time he runs a command (such as `p4 add *.*`), the rules are applied across the entire workspace.

To override (or ignore) the `P4IGNORE` file, use the `-I` option with the `p4 add`, `p4 reconcile`, or `p4 status` commands.

Changing files

To open a file for `edit`, issue the `p4 edit` command. When you open a file for edit, Perforce enables write permission for the file in your workspace and adds the file to a changelist. If the file is in the depot but not in your workspace, you must sync it before you open it for edit. You must open a file for edit before you attempt to edit the file.

Example 4.5. Opening a file for edit.

Bruno wants to make changes to `command.c`, so he syncs it and opens the file for edit.

```
p4 sync //depot/dev/command.c
//depot/dev/command.c#8 - added as c:\bruno_ws\dev\command.c

p4 edit //depot/dev/command.c
//depot/dev/command.c#8 - opened for edit
```

He then edits the file with any text editor. When he's finished, he submits the file to the depot with `p4 submit`, as described above.

Discarding changes (reverting)

To remove an open file from a changelist and discard any changes you made, issue the `p4 revert` command. When you revert a file, Perforce restores the last version you synced to your workspace. If you revert a file that is open for add, the file is removed from the changelist but is not deleted from your workspace.

Example 4.6. Reverting a file

Bruno decides not to add his text files after all.

```
p4 revert *.txt
//depot/dev/main/docs/manuals/installnotes.txt#none - was add, abandoned
//depot/dev/main/docs/manuals/requirements.txt#none - was add, abandoned
```

To preview the results of a revert operation without actually reverting files, specify the `-n` option when you issue the `p4 revert` command.

Deleting files

To delete files from the depot, you open them for delete by issuing the `p4 delete` command, then submit the changelist in which they are open. When you delete a file from the depot, previous revisions remain, and a new head revision is added, marked as "deleted." You can still sync previous revisions of the file.

When you issue the `p4 delete` command, the files are deleted from your workspace but not from the depot. If you revert files that are open for delete, they are restored to your workspace. When you successfully submit the changelist in which they are open, the files are deleted from the depot.

Example 4.7. Deleting a file from the depot.

Bruno deletes `vendor.doc` from the depot as follows:

```
p4 delete //depot/dev/main/docs/manuals/vendor.doc
//depot/dev/main/docs/manuals/vendor.doc#1 - opened for delete
```

The file is deleted from the client workspace immediately, but it is not deleted from the depot until he issues the `p4 submit` command.

Managing changelists

To change files in the depot, you open them in a *changelist*, make any changes to the files, and then *submit* the changelist. A changelist contains a list of files, their revision numbers, and the operations to be performed on the files. Unsubmitted changelists are referred to as *pending changelists*.

Submission of changelists is an all-or-nothing operation; that is, either all of the files in the changelist are updated in the depot, or, if an error occurs, none of them are. This approach guarantees that code alterations that affect multiple files occur simultaneously.

Perforce assigns numbers to changelists and also maintains a *default changelist*, which is numbered when you submit it. You can create multiple changelists to organize your work. For example, one changelist might contain files that are changed to implement a new feature, and another changelist might contain a bug fix. When you open a file, it is placed in the default changelist unless you specify an existing changelist number on the command line using the `-c` option. For example, to edit a file and submit it in changelist number 4, use `p4 edit -c 4` ***`filename`***. To open a file in the default changelist, omit the `-c` option.

You can also shelve changelists in order to temporarily preserve work in progress for your own use, or for review by others. Shelving enables you to temporarily cache files in the shared service without formally submitting them to the depot.

The Perforce service might renumber a changelist when you submit it, depending on other users' activities; if your changelist is renumbered, its original number is never reassigned to another changelist.

The commands that add or remove files from changelists are:

- `p4 add`
- `p4 delete`
- `p4 edit`

- `p4 integrate`
- `p4 reopen`
- `p4 revert`
- `p4 shelve`
- `p4 unshelve`

To submit a numbered changelist, specify the **-c** option when you issue the **p4 submit** command. To submit the default changelist, omit the **-c** option. For details, refer to the **p4 submit** command description in the *P4 Command Reference*.

To move files from one changelist to another, issue the **p4 reopen -c *changenum filenames*** command, where ***changenum*** specifies the number of the target changelist. If you are moving files to the default changelist, use `p4 reopen -c default` *`filenames`*.

Note	Using parallel submits can significantly improve performance. For additional information see the description of the **p4 submit** command in *P4 Command Reference*.

Creating numbered changelists

To create a numbered changelist, issue the **p4 change** command. This command displays the changelist form. Enter a description and make any desired changes; then save the form and exit the editor.

All files open in the default changelist are moved to the new changelist. When you exit the text editor, the changelist is assigned a number. If you delete files from this changelist, the files are moved back to the default changelist.

Example 4.8. Working with multiple changelists.

Bruno is fixing two different bugs, and needs to submit each fix in a separate changelist. He syncs the head revisions of the files for the first fix and opens the for edit in the default changelist:

```
C:\bruno_ws\>p4 sync //depot/dev/main/jam/*.c
[list of files synced...]

C:\bruno_ws>p4 edit //depot/dev/main/jam/*.c
[list of files opened for edit...]
```

Now he issues the **p4 change** command and enters a description in the changelist form. After he saves the file and exits the editor, Perforce creates a numbered changelist containing the files.

```
C:\bruno_ws\dev\main\docs\manuals>p4 change

    [Enter description and save form]

Change 777 created with 33 open file(s).
```

For the second bug fix, he performs the same steps, `p4 sync`, `p4 edit`, and `p4 change`. Now he has two numbered changelists, one for each fix.

The numbers assigned to submitted changelists reflect the order in which the changelists were submitted. When a changelist is submitted, Perforce might renumber it, as shown in the following example.

Example 4.9. Automatic renumbering of changelists

Bruno has finished fixing the bug that he's been using changelist 777 for. After he created that changelist, he submitted another changelist, and two other users also submitted changelists. Bruno submits changelist 777 with `p4 submit -c 777`, and sees the following message:

```
Change 777 renamed change 783 and submitted.
```

Submitting changelists

To submit a pending changelist, issue the `p4 submit` command. When you issue the `p4 submit` command, a form is displayed, listing the files in the changelist. You can remove files from this list. The files you remove remain open in the default pending changelist until you submit them or revert them.

To submit specific files that are open in the default changelist, issue the `p4 submit` *`filename`* command. To specify groups of files, use wildcards. For example, to submit all text files open in the default changelist, type `p4 submit "*".txt`. (Use quotation marks as an escape code around the `*` wildcard to prevent it from being interpreted by the local command shell).

After you save the changelist form and exit the text editor, the changelist is submitted to the Perforce service, and the files in the depot are updated. After a changelist has been successfully submitted, only a Perforce administrator can change it, and the only fields that can be changed are the description and user name.

If an error occurs when you submit the default changelist, Perforce creates a numbered changelist containing the files you attempted to submit. You must then fix the problems and submit the numbered changelist using the `-c` option.

Perforce enables write permission for files that you open for edit and disables write permission when you successfully submit the changelist containing the files. To prevent conflicts with Perforce's management of your workspace, do not change file write permissions manually.

Before committing a changelist, `p4 submit` briefly locks all files being submitted. If any file cannot be locked or submitted, the files are left open in a numbered pending changelist. By default, the files in a failed submit operation are left locked unless the `submit.unlocklocked` configurable is set. Files are unlocked even if they were manually locked prior to submit if submit fails when `submit.unlocklocked` is set.

Deleting changelists

To delete a pending changelist, you must first remove all files and jobs associated with it and then issue the `p4 change -d` *`changenum`* command. Related operations include the following:

- To move files to another changelist, issue the `p4 reopen -c` *`changenum`* command.

- To remove files from the changelist and discard any changes, issue the **p4 revert -c *changenum*** command.

Changelists that have already been submitted can be deleted only by a Perforce administrator. See the *Perforce Server Administrator's Guide: Fundamentals* for more information.

Renaming and moving files

To rename or move files, you must first open them for add or edit, and then use the **p4 move** command:

```
p4 move source_file target_file
```

To move groups of files, use matching wildcards in the ***source_file*** and ***target_file*** specifiers. To move files, you must have Perforce **write** permission for the specified files. For details about Perforce permissions, see the *Perforce Server Administrator's Guide: Fundamentals*.

When you rename or move a file using **p4 move**, the versioning service creates an integration record that links it to its deleted predecessor, preserving the file's history. Integration is also used to create branches and to propagate changes. For details, see "Integrating changes" on page 66.

Shelving work in progress

The Perforce shelving feature enables you to temporarily make copies of your files available to other users without checking the changelist into the depot.

Shelving is useful for individual developers who are switching between tasks or performing cross-platform testing before checking in their changes. Shelving also enables teams to easily hand off changes and to perform code reviews.

Example 4.10. Shelving a changelist.

Earl has made changes to **command.c** on a UNIX platform, and now wants others to be able to view and test his changes.

```
$ p4 edit //depot/dev/command.c
//depot/dev/command.c#9 - opened for edit
...

$ p4 shelve
Change 123 created with 1 open file(s).
Shelving files for change 123.
edit //depot/dev/command.c#9
Change 123 files shelved.
```

A pending changelist is created, and the shelved version of **command.c** is stored in the service. The file **command.c** remains editable in Earl's workspace, and Earl can continue to work on the file, or can revert his changes and work on something else.

Shelved files remain open in the changelist from which they were shelved. (To add a file to an existing shelved changelist, you must first open that file in that specific changelist.) You can continue to work on the files in your workspace without affecting the shelved files. Shelved files can be synced to other workspaces, including workspaces owned by other users. For example:

Example 4.11. Unshelving a changelist for code review

Earl has asked for code review and a cross-platform compatibility check on the version of **command.c** that he shelved in changelist 123. Bruno, who is using a Windows machine, types:

```
C:\bruno_ws\dev> p4 unshelve -s 123 //depot/dev/command.c
//depot/dev/command.c#9 - unshelved, opened for edit
```

and conducts the test in the Windows environment while Earl continues on with other work.

When you shelve a file, the version on the shelf is unaffected by commands that you perform in your own workspace, even if you revert the file to work on something else.

Example 4.12. Handing off files to other users.

Earl's version of **command.c** works on UNIX, but Bruno's cross-platform check of **command.c** has revealed a bug. Bruno can take over the work from here, so Earl reverts his workspace and works on something else:

```
$ p4 revert //depot/dev/command.c
//depot/dev/command.c#9 - was edit, reverted
```

The shelved version of **command.c** is still available from Earl's pending changelist 123, and Bruno opens it in a new changelist, changelist 124.

```
$ p4 unshelve -s 123 -c 124 //depot/dev/command.c
//depot/dev/command.c#9 - unshelved, opened for edit
```

When Bruno is finished with the work, he can either re-shelve the file (in his own changelist 124, not Earl's changelist 123) for further review, or discard the shelved file and submit the version in his workspace by using **p4 submit**.

The **p4 submit** command has a **-e** option that enables the submitting of shelved files directly from a changelist. All files in the shelved change must be up to date and resolved. Other restrictions can apply in the case of files shelved to stream targets; see the *P4 Command Reference* for details. (To avoid dealing with these restrictions, you can always move the shelved files into a new pending changelist before submitting that changelist.)

Example 4.13. Discarding shelved files before submitting a change.

The Windows cross-platform changes are complete, and changelist 124 is ready to be submitted. Bruno uses **p4 shelve -d** to discard the shelved files.

```
C:\bruno_ws\dev> p4 shelve -d -c 124
Shelve 124 deleted.
```

All files in the shelved changelist are deleted. Bruno can now submit the changelist.

```
C:\bruno_ws\dev> p4 submit -c 124
Change 124 submitted.
```

Bruno could have shelved the file in changelist 124, and let Earl unshelve it back into his original changelist 123 to complete the check-in.

Displaying information about changelists

To display brief information about changelists, use the `p4 changes` command. To display full information, use the `p4 describe` command. The following table describes some useful reporting commands and options.

Command	Description
`p4 changes`	Displays a list of all pending, submitted, and shelved changelists, one line per changelist, and an abbreviated description.
`p4 changes -m` *`count`*	Limits the number of changelists reported on to the last specified number of changelists.
`p4 changes -s` *`status`*	Limits the list to those changelists with a particular status; for example, `p4 changes -s submitted` lists only already submitted changelists.
`p4 changes -u` *`user`*	Limits the list to those changelists submitted by a particular user.
`p4 changes -c` *`workspace`*	Limits the list to those changelists submitted from a particular client workspace.
`p4 describe` *`changenum`*	Displays full information about a single changelist. If the changelist has already been submitted, the report includes a list of affected files and the diffs of these files. (You can use the `-s` option to exclude the file diffs.)
`p4 describe -O` *`changenum`*	If a changelist was renumbered, describe the changelist in terms of its original change number. (For example, the changelist renumbered in the example on Example 4.9, "Automatic renumbering of changelists" on page 44 can be retrieved with either `p4 describe 783` or `p4 describe -O 777`.)

For more information, see "Changelist reporting" on page 97.

Diffing files

Perforce provides the ability to *diff* (compare) revisions of text files. By diffing files, you can display:

- Changes that you made after opening the file for edit

- Differences between any two revisions
- Differences between file revisions in different branches

To diff a file that is synced to your workspace with a depot revision, issue the `p4 diff` *`filename#rev`* command. If you omit the revision specifier, the file in your workspace is compared with the revision you last synced, to display changes you made after syncing it.

To diff two revisions that reside in the depot but not in your workspace, use the `p4 diff2` command. To diff a set of files, specify wildcards in the filename argument when you issue the `p4 diff2` command.

The `p4 diff` command performs the comparison on your workstation, but the `p4 diff2` command instructs the Perforce service to perform the diff and to send the results to you.

The following table lists some common uses for diff commands.

To diff	Against	Use this command
The workspace file	The head revision	`p4 diff` *`file`* or `p4 diff` *`file`*`#head`
The workspace file	Revision 3	`p4 diff` *`file`*`#3`
The head revision	Revision 134	`p4 diff2` *`file file`*`#134`
File revision at changelist 32	File revision at changelist 177	`p4 diff2` *`file`*`@32` *`file`*`@177`
The workspace file	A file shelved in pending changelist 123	`p4 diff file@=123`
All files in release 1	All files in release 2	`p4 diff2 //depot/rel1/... //depot/rel2/...`

By default, the `p4 diff` command launches Perforce's internal diff application. To use a different diff program, set the `P4DIFF` environment variable to specify the path and executable of the desired application. To specify arguments for the external diff application, use the `-d` option. For details, refer to the *P4 Command Reference*.

Working offline

The preferred method of working offline (without access to the Perforce service) is to use the Perforce Sandbox. For details, refer to the *Perforce Sandbox User's Guide*.

If you are not using the Perforce Sandbox, you must manually reconcile your work with the Perforce service when you regain access to it. The following method for working detached assumes that you work on files in your workspace or update the workspace with your additions, changes, and deletions before you update the depot.

To work offline:

1. Work on files without issuing `p4` commands. Instead, use operating system commands to change the permissions on files.
2. After the network connection is reestablished, use `p4 status` or `p4 reconcile` to find all files in your workspace that have changed.
3. Submit the resulting changelist(s).

To detect changed files, issue the `p4 status` or `p4 reconcile` commands. The commands perform essentially the same function, but differ in their default behavior and output format.

Command	Description
`p4 reconcile`	When called without arguments, `p4 reconcile` opens the files in a changelist. To preview an operation, you must either use the `-n` option with `p4 reconcile`, or use the `p4 status` command.
`p4 status`	When called without arguments, `p4 status` only previews the results of the workspace reconciliation. You must use either `p4 status -A` (or some combination of the `-e`, `-a`, or `-d` options) to actually open the files in a changelist.

Chapter 5

Resolving Conflicts

This chapter tells you how to work in a team development environment, where multiple users who are working on the same files might need to reconcile their changes.

In settings where multiple users are working on the same set of files, conflicts can occur. Perforce enables your team to work on the same files simultaneously and resolve any conflicts that arise. For example, conflicts occur if two users change the same file (the primary concern in team settings) or you edit a previous revision of a file rather than the head revision.

When you attempt to submit a file that conflicts with the head revision in the depot, Perforce requires you to resolve the conflict. Merging changes from a development branch to a release branch is another typical task that requires you to resolve files.

To prevent conflicts, Perforce enables you to lock files when they are edited. However, locking can restrict team development. Your team needs to choose the strategy that maximizes file availability while minimizing conflicts. For details, refer to "Locking files" on page 59.

You might prefer to resolve files using graphical tools like P4V, the Perforce Visual Client, and its associated visual merge tool P4Merge.

How conflicts occur

File conflicts can occur when two users edit and submit two versions of the same file. Conflicts can occur in a number of ways, for example:

1. Bruno opens `//depot/dev/main/jam/command.c#8` for edit.
2. Gale subsequently opens the same file for edit in her own client workspace.
3. Bruno and Gale both edit `//depot/dev/main/jam/command.c#8`.
4. Bruno submits a changelist containing `//depot/dev/main/jam/command.c`, and the submit succeeds.
5. Gale submits a changelist with her version of `//depot/dev/main/jam/command.c`. Her submit fails.

If Perforce accepts Gale's version into the depot, her changes will overwrite Bruno's changes. To prevent Bruno's changes from being lost, Perforce rejects the changelist and schedules the conflicting file to be resolved. If you know of file conflicts in advance and want to schedule a file for resolution, sync it. Perforce detects the conflicts and schedules the file for resolution.

How to resolve conflicts

To resolve a file conflict, you determine the contents of the files you intend to submit by issuing the `p4 resolve` command and choosing the desired method of resolution for each file. After you resolve conflicts, you submit the changelist containing the files.

Note	If you open a file for edit, then sync a subsequently submitted revision from the depot, Perforce requires you to resolve to prevent your own changes from being overwritten by the depot file.

By default, Perforce uses its diff program to detect conflicts. You can configure a third-party diff program. For details, see "Diffing files" on page 47.

To resolve conflicts and submit your changes, perform the following steps:

1. Sync the files (for example `p4 sync //depot/dev/main/jam/...`). Perforce detects any conflicts and schedules the conflicting files for resolve.
2. Issue the `p4 resolve` command and resolve any conflicts. See "Options for resolving conflicts" on page 52 for details about resolve options.
3. Test the resulting files (for example, compile code and verify that it runs).
4. Submit the changelist containing the files.

Note

If any of the three file revisions participating in the merge are binary instead of text, a three-way merge is not possible. Instead, `p4 resolve` performs a two-way merge: the two conflicting file versions are presented, and you can choose between them or edit the one in your workspace before submitting the changelist.

Your, theirs, base and merge files

The `p4 resolve` command uses the following terms during the merge process.

File revision	Description
yours	The revision of the file in your client workspace, containing changes you made.
theirs	The revision in the depot, edited by another user, that **yours** conflicts with. (Usually the head revision, but you can schedule a resolve with another revision using `p4 sync`.)
base	The file revision in the depot that ***yours*** and theirs were edited from (the closest common ancestor file).
merge	The file generated by Perforce from ***theirs***, ***yours***, and ***base***.
result	The final file resulting from the resolve process.

Options for resolving conflicts

To specify how a conflict is to be resolved, you issue the `p4 resolve` command, which displays a dialog for each file scheduled for resolve. The dialog describes the differences between the file you changed and the conflicting revision. For example:

```
p4 resolve //depot/dev/main/jam/command.c
c:\bruno_ws\dev\main\jam\command.c - merging //depot/dev/main/jam/command.c#9

Diff chunks: 4 yours + 2 theirs + 1 both + 1 conflicting
Accept(a) Edit(e) Diff(d) Merge (m) Skip(s) Help(?) e:
```

The differences between each pair of files are summarized by `p4 resolve`. Groups of lines (chunks) in the `yours`, `theirs`, and `base` files can differ in various ways. Chunks can be:

- **Diffs**: different between two of the three files: ***`yours`***, ***`theirs`***, or ***`base`***
- **Conflicts**: different in all three files

In the preceding example:

- Four chunks are identical in ***`theirs`*** and ***`base`*** but are different in ***`yours`***.
- Two chunks are identical in ***`yours`*** and ***`base`*** but are different in ***`theirs`***.
- One chunk was changed identically in ***`yours`*** and ***`theirs`***.
- One chunk is different in ***`yours`***, ***`theirs`***, and ***`base`***.

Perforce's recommended choice is displayed at the end of the command line. Pressing **ENTER** or choosing **Accept** performs the recommended choice.

You can resolve conflicts in three basic ways:

- Accept a file without changing it (see "Accepting yours, theirs, or merge" on page 53)
- Edit the merge file with a text editor (see "Editing the merge file" on page 54)
- Merge changes selectively using a merge program (see "Merging to resolve conflicts" on page 54)

The preceding options are interactive. You can also specify resolve options on the `p4 resolve` command line, if you know which file you want to accept. For details, see "Resolve command-line options" on page 58. To reresolve a resolved but unsubmitted file, specify the `-f` option when you issue the `p4 resolve` command. You cannot reresolve a file after you submit it. The following sections describe the resolve options in more detail.

Accepting yours, theirs, or merge

To accept a file without changing it, specify one of the following options.

Option	Description	Remarks
a	Accept recommended file	• If ***`theirs`*** is identical to ***`base`***, accept ***`yours`***. • If ***`yours`*** is identical to ***`base`***, accept ***`theirs`***. • If ***`yours`*** and ***`theirs`*** are different from ***`base`***, and there are no conflicts between ***`yours`*** and ***`theirs`***; accept ***`merge`***. • Otherwise, there are conflicts between ***`yours`*** and ***`theirs`***, so skip this file.

Option	Description	Remarks
`ae`	Accept edit	If you edited the ***merge*** file (by selecting `e` from the `p4 resolve` dialog), accept the edited version into the client workspace. The version in the client workspace is overwritten.
`am`	Accept ***merge***	Accept ***merge*** into the client workspace as the resolved revision. The version in the client workspace is overwritten.
`at`	Accept ***theirs***	Accept ***theirs*** into the client workspace as the resolved revision. The version in the client workspace is overwritten.
`ay`	Accept ***yours***	Accept ***yours*** into the client workspace as the resolved revision, ignoring changes that might have been made in ***theirs***.

Accepting `yours`, `theirs`, `edit`, or `merge` overwrites changes, and the generated merge file might not be precisely what you want to submit to the depot. The most precise way to ensure that you submit only the desired changes is to use a merge program or edit the merge file.

Editing the merge file

To resolve files by editing the merge file, choose the `e` option. Perforce launches your default text editor, displaying the merge file. In the merge file, diffs and conflicts appear in the following format:

```
>>>> ORIGINAL file#n
(text from the original version)
==== THEIR file#m
(text from their file)
==== YOURS file
(text from your file)
<<<<
```

To locate conflicts and differences, look for the difference marker "`>>>>`" and edit that portion of the text. Examine the changes made to ***theirs*** to make sure that they are compatible with your changes. Make sure you remove all conflict markers before saving. After you make the desired changes, save the file. At the `p4 resolve` prompt, choose `ae`.

By default, only the conflicts between the ***yours*** and ***theirs*** files are marked. To generate difference markers for all differences, specify the `-v` option when you issue the `p4 resolve` command.

Merging to resolve conflicts

A merge program displays the differences between yours, theirs, and the base file, and enables you to select and edit changes to produce the desired result file. To configure a merge program, set `P4MERGE` to the desired program. To use the merge program during a resolve, choose the `m` option. For details about using a specific merge program, consult its online help.

After you merge, save your results and exit the merge program. At the `p4 resolve` prompt, choose `am`.

Full list of resolve options

The `p4 resolve` command offers the following options.

Option	Action	Remarks
`?`	Help	Display help for `p4 resolve`.
`a`	Accept automatically	Accept the autoselected file: • If ***theirs*** is identical to ***base***, accept ***yours***. • If ***yours*** is identical to ***base***, accept ***theirs***. • If ***yours*** and ***theirs*** are different from ***base***, and there are no conflicts between ***yours*** and ***theirs***; accept ***merge***. • Otherwise, there are conflicts between ***yours*** and ***theirs***, so skip this file.
`ae`	Accept edit	If you edited the ***merge*** file (by selecting `e` from the `p4 resolve` dialog), accept the edited version into the client workspace. The version in the client workspace is overwritten.
`am`	Accept ***merge***	Accept ***merge*** into the client workspace as the resolved revision. The version in the client workspace is overwritten.
`at`	Accept ***theirs***	Accept ***theirs*** into the client workspace as the resolved revision. The version in the client workspace is overwritten.
`ay`	Accept ***yours***	Accept ***yours*** into the client workspace as the resolved revision, ignoring changes that might have been made in ***theirs***.
`d`	Diff	Show diffs between ***merge*** and ***yours***.
`dm`	Diff ***merge***	Show diffs between ***merge*** and ***base***.
`dt`	Diff ***theirs***	Show diffs between ***theirs*** and ***base***.
`dy`	Diff ***yours***	Show diffs between ***yours*** and ***base***.
`e`	Edit merged	Edit the preliminary merge file generated by Perforce.
`et`	Edit ***theirs***	Edit the revision in the depot that the client revision conflicts with (usually the head revision). This edit is read-only.
`ey`	Edit ***yours***	Edit the revision of the file currently in the workspace.
`m`	Merge	Invoke the command `P4MERGE` ***base theirs yours merge***. To use this option, you must set `P4MERGE` to the name of a third-party program that merges the first three files and writes the fourth as a result.

Option	Action	Remarks
s	Skip	Skip this file and leave it scheduled for resolve.

Note The ***merge*** file is generated by the Perforce service, but the differences displayed by **dy**, **dt**, **dm**, and **d** are generated by your workstation's diff program. To configure another diff program to be launched when you choose a **d** option during a resolve, set **P4DIFF**. For more details, see "Diffing files" on page 47.

Example 5.1. Resolving file conflicts

To resolve conflicts between his work on a Jam readme file and Earl's work on the same file, Bruno types `p4 resolve //depot/dev/main/jam/README` and sees the following:

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

Bruno sees that that he and Earl have made a conflicting change to the file. He types **e** to edit the merge file and searches for the difference marker "**>>>>**". The following text is displayed:

```
Jam/MR (formerly "jam - make(1) redux")
/+\
>>>> ORIGINAL README#26
    +\ Copyright 1993, 1997 Christopher Seiwald.
==== THEIRS README#27
    +\ Copyright 1993, 1997, 2004 Christopher Seiwald.
==== YOURS README
    +\ Copyright 1993, 1997, 2005 Christopher Seiwald.
<<<<
    \+/
```

Bruno and Earl have updated the copyright date differently. Bruno edits the merge file so that the header is correct, exits from the editor and types **am**. The edited merge file is written to the client workspace, and he proceeds to resolve the next file.

When a version of the file is accepted during a resolve, the file in the workspace is overwritten, and the new client file must still be submitted to the depot. New conflicts can occur if new versions of a file are submitted after you resolve but before you submit the resolved files. This problem can be prevented by locking the file before you perform the resolve. For details, see "Locking files" on page 59.

Resolving Branched Files, Deletions, Moves and Filetype Changes

Beyond reconciling changes to the contents of related files after integration, you can also determine how other kinds of changes are handled. For example:

- You edit **header.cc** in the mainline while a coworker deletes it in the release branch (or vice versa). You integrate fixes in the release branch back to main. During resolve, you can decide whether `header.cc` is deleted from the mainline or the action in the release branch is ignored, preserving `header.cc` in the mainline.

- A developer implement RCS keywords in source files in a development branch, and change their Perforce filetype from `text` to `text+k`. The release manager wants to integrate new features from the development branch to the mainline, but does not want to enable keyword expansion in the mainline. During resolve, the release manager can choose to ignore the filetype change.
- The file `header.cc` is branched from `main` to `rel`. Subsequently, it's renamed to `headerx.cc` in main, and moved in the release branch to the `headers` subfolder.

Following are simple cases describing how you can resolve non-content changes to related files. After a source file is branched to a target file, changes are made as describe below, then you integrate the source to the target. To choose the outcome, you specify the resolve options `at` ("Accept Theirs") or `ay` ("Accept Yours") as follows:

- **The source is edited and target is deleted**: the `at` option re-adds the source in the target branch. The `ay` option causes the file to remain deleted in the target branch.
- **The source is deleted and the target is edited**: the `at` option causes the file to be deleted in the target branch. The `ay` option retains the edited content in the target branch.
- **The target file was moved after being branched**: the `at` option moves the target file to the source file name and location. The `ay` option retains the target file name and location.
- **The filetype of the source file was changed after it was branched**: the `at` option propagates the change to the target. The `ay` option leaves the filetype of the target unchanged. If the differing filetypes do not conflict, you have the option of combining them.
- **Files have been moved or renamed in conflicting ways**: you are prompted to choose a path and filename. Example:

```
Resolving move to //depot/rel/headerx.cc
Filename resolve:
at: //depot/rel/headerx.cc
ay: //depot/rel/headers/header.cc
am: //depot/rel/headers/headerx.cc
```

By default, the `p4 resolve` command resolves all types of change, content and non-content. To constrain the type of actions that you want to resolve, specify the `-A` option as follows:

Option	What is Resolved
`-Aa`	Resolve attributes set by `p4 attribute`.
`-Ab`	Integrations where the source is edited and the target is deleted.
`-Ac`	Resolve file content changes as well as actions.
`-Ad`	Integrations where the source is deleted and target is edited.

Option	What is Resolved
`-Am`	Renames and moves.
`-At`	Filetype changes.
`-AQ`	Charset changes.

To perform more than one type of resolve, combine the options (for example: `-Abd`). By default, resolving is performed file by file, interactively. To specify the same outcome for a particular action (for example, propagate all moves), and avoid the prompting, include the desired option on the command line. For example: `p4 resolve -Am -at`

Resolve command-line options

The following `p4 resolve` options enable you to resolve directly instead of interactively. When you specify one of these options in the `p4 resolve` command, files are resolved as described in the following table.

Option	Description
`-a`	Accept the autoselected file.
`-ay`	Accept ***yours***.
`-at`	Accept ***theirs***. Use this option with caution, because the file revision in your client workspace is overwritten with the head revision from the depot, and you cannot recover your changes.
`-am`	Accept the recommended file revision according to the following logic: • If ***theirs*** is identical to ***base***, accept ***yours***. • If ***yours*** is identical to ***base***, accept ***theirs***. • If ***yours*** and ***theirs*** are different from ***base***, and there are no conflicts between ***yours*** and ***theirs***, accept ***merge***. • Otherwise, there are conflicts between ***yours*** and ***theirs***, so skip this file, leaving it unresolved.
`-af`	Accept the recommended file revision, even if conflicts remain. If this option is used, edit the resulting file in the workspace to remove any difference markers.
`-as`	Accept the recommended file revision according to the following logic: • If ***theirs*** is identical to ***base***, accept ***yours***. • If ***yours*** is identical to ***base***, accept ***theirs***. • Otherwise skip this file.

Example 5.2. Automatically accepting particular revisions of conflicting files

Bruno has been editing the documentation files in `/doc` and knows that some of them require resolving. He types `p4 sync doc/*.guide`, and all of these files that conflict with files in the depot are scheduled for resolve.

He then types `p4 resolve -am` and the merge files for all scheduled resolves are generated, and those merge files that contain no line set conflicts are written to his client workspace. He'll still need to manually resolve any conflicting files, but the amount of work he needs to do is substantially reduced.

Resolve reporting commands

The following reporting commands are helpful when you are resolving file conflicts.

Command	Meaning
`p4 diff [`*filenames*`]`	Diffs the file revision in the workspace with the last revision you synced, to display changes you have made.
`p4 diff2` *file1 file2*	Diffs two depot files. The specified files can be any two file revisions and different files. When you diff depot files, Perforce service uses its own diff program, not the diff program configured by setting `P4DIFF`.
`p4 sync -n [`*filenames*`]`	Previews the specified sync, listing which files have conflicts and need to be resolved.
`p4 resolved`	Reports files that have been resolved but not yet submitted.

Locking files

After you open a file, you can lock it to prevent other users from submitting it before you do. The benefit of locking a file is that conflicts are prevented, but when you lock a file, you might prevent other team members from proceeding with their work on that file.

Preventing multiple resolves by locking files

Without file locking, there is no guarantee that the resolve process ever ends. The following scenario demonstrates the problem:

1. Bruno opens file for edit.
2. Gale opens the same file in her client for edit.
3. Bruno and Gale both edit their client workspace versions of the file.
4. Bruno submits a changelist containing that file, and his submit succeeds.
5. Gale submits a changelist with her version of the file; her submit fails because of file conflicts with the new depot's file.

6. Gale starts a resolve.
7. Bruno edits and submits a new version of the same file.
8. Gale finishes the resolve and attempts to submit; the submit fails and must now be merged with Bruno's latest file.

   …and so on.

To prevent such problems, you can lock files, as follows.

1. Before scheduling a resolve, lock the file.
2. Sync the file (to schedule a resolve).
3. Resolve the file.
4. Submit the file.
5. Perforce automatically unlocks the file after successful changelist submission.

To list open locked files on UNIX, issue the following command:

```
p4 opened | grep "*locked*"
```

Preventing multiple checkouts

To ensure that only one user at a time can work on the file, use the `+l` (exclusive-open) file type modifier. For example:

```
p4 reopen -t binary+l file
```

Although exclusive locking prevents concurrent development, for some file types (binary files), merging and resolving are not meaningful, so you can prevent conflicts by preventing multiple users from working on the file simultaneously.

Your Perforce administrator can use the `p4 typemap` command to ensure that all files of a specified type (for instance, `//depot/.../*.gif` for all `.gif` files) can only be opened by one user at a time. See the *P4 Command Reference* for details.

The difference between `p4 lock` and `+l` is that `p4 lock` allows anyone to open a file for edit, but only the person who locked the file can submit it. By contrast, a file of type `+l` prevents more than one user from opening the file.

Chapter 6 Codelines, Branching and Streams

This chapter describes the tasks required to maintain groups of files in your depot. The following specific issues are addressed:

- Depot directory structure and how to best organize your repository
- Moving files and file changes among codeline and project directories
- Identifying specific sets of files using either labels or changelists

To make codeline management easier, you can use *streams*, a Perforce feature that encapsulates numerous best practices and automations. This chapter includes a section describing how you use the Perforce command line client to manage streams. For basic information, refer to *Introducing Perforce*. For more details about using streams, see the *P4V User's Guide*, "Working with Streams".

This chapter focuses on maintaining a software code base, but many of the tasks are relevant to managing other groups of files, such as a web site. For advice about best practices, see the white papers on the Perforce web site.

Basic terminology

To enable you to understand the following sections, here are definitions of some relevant terms as they are used in Perforce.

Term	Definition
branch	(*noun*) A set of related files created by copying files, as opposed to adding files. A group of related files is often referred to as a *codeline*. (*verb*) To create a branch.
integrate	To create new files from existing files, preserving their ancestry (branching), or to propagate changes from one set of files to another (merging).
merge	The process of combining the contents of two conflicting file revisions into a single file, typically using a merge tool like P4Merge.
resolve	The process you use to reconcile the differences between two revisions of a file. You can choose to resolve conflicts by selecting a file to be submitted or by merging the contents of conflicting files.

Organizing the depot

You can think of a depot as a top-level directory. Consider the following factors as you decide how to organize your depot:

- **Type of content**: create depots or mainline directories according to the nature of your projects and their relationships (for example, applications with multiple components developed on separate schedules).

- **Release requirements**: within a project, create branches for each release and integrate changes between branches to control the introduction of features and bug fixes.
- **Build management**: use labels and changelists to control the file revisions that are built; use client specifications and views to ensure clean build areas.

A basic and logical way to organize the depot is to create one subdirectory (codeline) for each project. For example, if your company is working on Jam, you might devote one codeline to the release presently in development, another to already-released software, and perhaps one to your corporate web site. Your developers can modify their client views to map the files in their project, excluding other projects that are not of interest. For example, if Earl maintains the web site, his client view might look like this:

```
//depot/www/dev/...     //earl-web-catalpa/www/development/...
//depot/www/review/...  //earl-web-catalpa/www/review/...
//depot/www/live/...    //earl-web-catalpa/www/live/...
```

And Gale, who's working on Jam, sets up her client view as:

```
//depot/dev/main/jam/... //gale-jam-oak/jam/...
```

You can organize according to projects or according to the purpose of a codeline. For example, to organize the depot according to projects, you can use a structure like the following:

```
//depot/project1/main/
//depot/project1/release 1.0/
//depot/project1/release 1.1/
```

Or, to organize the depot according to the purpose of each codeline, you can use a structure like the following:

```
//depot/main/project1/
//depot/main/project2/
//depot/release1.0/project1/
//depot/release1.0/project2/
//depot/release2.0/project1/
//depot/release2.0/project2/
```

Another approach is to create one depot for each project. Choose a structure that makes branching and integrating as simple as possible, so that the history of your activities makes sense to you.

Populating Codelines

If you are creating a codeline that has no history, use the `p4 add` command to add files to it, then use `p4 copy` to create branches. For example, to create the mainline structure shown in the previous section, perform the following steps:

1. Create a local folder your workspace for the mainline files; for example:

```
mkdir c:\p4clients\myworkspace\depot\main\
```

2. Copy the files for Project1 and Project2 to the newly created folder.
3. Add the files to the depot:

```
p4 add //depot/main/project1/...
p4 add //depot/main/project2/...
p4 submit
```

4. Create release branches:

```
p4 copy //depot/main/project1/... //depot/release1.0/project1/...
p4 copy //depot/main/project2/... //depot/release1.0/project2/...
p4 submit
```

Now you can use the **p4 copy**, **p4 merge** and **p4 integrate** commands to propagate changes between main and release branches. (You can also seed a codeline from another codeline using the **p4 integrate** command, if there is a historical relationship between the source and target that you need to preserve.)

A shortcut: p4 populate

If a target codeline is completely empty (no files present, not even deleted files), Perforce offers a command that automates the process of copying the files from an existing source codeline submitting the associated changelist.

For example, instead of populating a **release1.0** branch with the following two commands:

```
p4 copy //depot/main/project1/... //depot/release1.0/project1/...
p4 submit
```

you can use the **p4 populate** command to populate the branch:

```
p4 populate //depot/main/project1/... //depot/release1.0/project1/...
```

Branching Codelines

Branching is a method of maintaining the relationship between sets of related files. Branches can evolve separately from their ancestors and descendants, and you can propagate (*integrate*) changes from one branch to another as desired. Perforce's *Inter-File Branching*™ mechanism preserves the relationship between files and their ancestors while consuming minimal resources.

To create a branch, use the `p4 integrate` command. The `p4 integrate` command is also used to propagate changes between existing sets of files. For details about integrating changes, refer to "Integrating changes" on page 66.

When to branch

Create a branch when two sets of files have different submission policies or need to evolve separately. For example:

- *Problem* : the development group wants to submit code to the depot whenever their code changes, regardless of whether it compiles, but the release engineers don't want code to be submitted until it's been debugged, verified, and approved.

  Solution: create a release branch by branching the development codeline. When the development codeline is ready, it is integrated into the release codeline. Patches and bug fixes are made in the release code and integrated back into the development code.

- *Problem*: a company is writing a driver for a new multiplatform printer. The UNIX device driver is done and they are beginning work on an OS X driver, using the UNIX code as their starting point.

  Solution: create an OS X branch from the existing UNIX code. These two codelines can evolve separately. If bugs are found in one codeline, fixes can be integrated to the other.

One basic strategy is to develop code in `//depot/main/` and create branches for releases (for example, `//depot/rel1.1/`). Make release-specific bug fixes in the release branches and, if required, integrate them back into the `//depot/main/` codeline.

Creating branches

To create a branch, use the `p4 integrate` command. When you create a branch, Perforce records the relationships between the branched files and their ancestors.

You can create branches using file specifications or branch specifications. For simple branches, use file specifications. For branches that are based on complex sets of files or to ensure that you have a record of the way you defined the branch, use branch specifications. Branch specifications can also be used in subsequent integrations. Branch specifications also can serve as a record of codeline policy.

Using branch specifications

To map a set of files from source to target, you can create a *branch mapping* and use it as an argument when you issue the `p4 integrate` command. To create a branch mapping, issue the `p4 branch` ***`branchname`*** command and specify the desired mapping in the `View:` field, with source files on the left and target files on the right. Make sure that the target files and directories are in your client view. Creating or altering a branch mapping has no effect on any files in the depot or client workspace. The branch mapping merely maps source files to target files.

To use the branch mapping to create a branch, issue the `p4 integrate -b` ***`branchname`*** command; then use `p4 submit` to submit the target files to the depot.

Branch specifications can contain multiple mappings and exclusionary mappings, just as client views can. For example, the following branch mapping branches the `Jam 1.0` source code, excluding test scripts, from the main codeline.

```
Branch:   jamgraph-1.0-dev2release

View:
    //depot/dev/main/jamgraph/...       //depot/release/jamgraph/1.0/...
    -//depot/dev/main/jamgraph/test/... //depot/release/jamgraph/1.0/test/...
    //depot/dev/main/bin/glut32.dll     //depot/release/jamgraph/1.0/bin/glut32.dll
```

To create a branch using the preceding branch mapping, issue the following command:

```
p4 integrate -b jamgraph-1.0-dev2release
```

and use `p4 submit` to submit the changes.

To delete a branch mapping, issue the `p4 branch -d` *`branchname`* command. Deleting a branch mapping has no effect on existing files or branches.

As with workspace views, if a filename or path in a branch view contains spaces, make sure to quote the path:

```
//depot/dev/main/jamgraph/... "//depot/release/Jamgraph 1.0/..."
```

Using file specifications

To branch using file specifications, issue the `p4 integrate` command, specifying the source files and target files. The target files must be in the client view. If the source files are not in your client view, specify them using depot syntax.

To create a branch using file specifications, perform the following steps:

1. Determine where you want the branch to reside in the depot and the client workspace. Add the corresponding mapping specification to your client view.
2. Issue the `p4 integrate` *`source_files target_files`* command.
3. Submit the changelist containing the branched files. The branch containing the target files is created in the depot.

Example 6.1. Creating a branch using a file specification

Version 2.2 of Jam has just been released, and work on version 3.0 is starting. Version 2.2 must be branched to `//depot/release/jam/2.2/...` for maintenance.

Bruno uses `p4 client` to add the following mapping to his client view:

```
//depot/release/jam/2.2/... //bruno_ws/release/jam/2.2/...
```

He issues the following command to create the branch:

```
p4 integrate //depot/dev/main/jam/... //bruno_ws/release/jam/2.2/...
```

Finally, he issues the `p4 submit` command, which adds the newly branched files to the depot.

Integrating changes

After you create branches, you might need to propagate changes between them. For example, if you fix a bug in a release branch, you probably want to incorporate the fix back into your main codeline. To propagate selected changes between branched files, you use the **p4 integrate**, **p4 merge**, or **p4 copy** commands, as follows:

1. Issue the **p4 integrate** command to schedule the files for resolve. (In many cases, you can also use **p4 merge** or **p4 copy**.)
2. Issue the **p4 resolve** command to propagate changes from the source files to the target files.

   To propagate individual changes, edit the merge file or use a merge program. The changes are made to the target files in the client workspace.
3. Submit the changelist containing the resolved files.

Example 6.2. Propagating changes between branched files

Bruno has fixed a bug in the release 2.2 branch of the Jam project and needs to integrate it back to the main codeline. From his home directory, Bruno types:

p4 integrate //depot/release/jam/2.2/src/Jambase //depot/dev/main/jam/Jambase

and sees the following message:

```
//depot/dev/main/jam/Jambase#134 - integrate from //depot/release/jam/2.2/src/Jambase#9
```

The file has been scheduled for resolve. He types **p4 resolve**, and the standard merge dialog appears on his screen.

```
//depot/dev/main/jam/Jambase - merging depot/release/jam/2.2/src/Jambase#9
Diff chunks: 0 yours + 1 theirs + 0 both + 0 conflicting
Accept(a) Edit(e) Diff(d) Merge (m) Skip(s) Help(?) [at]:
```

He resolves the conflict. When he's done, the result file overwrites the file in his workspace. The changelist containing the file must be submitted to the depot.

To run the **p4 integrate**, **p4 merge**, or **p4 copy** commands, you must have Perforce **write** permission on the target files, and **read** access on the source files. (See the *Perforce Server Administrator's Guide: Fundamentals* for information on Perforce permissions.)

By default, a file that has been newly created in a client workspace by **p4 integrate** cannot be edited before being submitted. To edit a newly integrated file before submission, resolve it, then issue the **p4 edit** command.

If the range of revisions being integrated includes deleted revisions (for example, a file was deleted from the depot, then re-added), you can specify how deleted revisions are integrated using the **-Di** option. For details, refer to the *P4 Command Reference*.

Integrating using branch specifications

To integrate changes from one set of files and directories to another, you can use a branch mapping when you issue the `p4 integrate` command. The basic syntax of the integrate command when using a branch mapping is:

`p4 integrate -b` ***`branchname`*** `[`*`tofiles`*`]`

Target files must be mapped in both the branch view and the client view. The source files need not be in the client view. If you omit the ***`tofiles`*** argument, all the files in the branch are affected.

To reverse the direction of integration using a branch mapping, specify the `-r` option. This option enables you to integrate in either direction between two branches without requiring you to create a branch mapping for each direction.

Example 6.3. Integrating changes to a single file in a branch

A feature has been added in the main Jam codeline and Bruno wants to propagate the feature to release 1.0. He types:

```
p4 integrate -b jamgraph-1.0-dev2release *.c
```

and sees:

```
//depot/release/jam/1.0/src/command.c#10 - integrate from //depot/dev/main/jam/command.c#97
```

The file has been scheduled for resolve. He types `p4 resolve`, and the standard merge dialog appears on his screen.

```
//depot/release/jam/1.0/src/command.c - merging //depot/dev/main/jam/command.c#97
Diff chunks: 0 yours + 1 theirs + 0 both + 0 conflicting
Accept(a) Edit(e) Diff(d) Merge (m) Skip(s) Help(?) [at]:
```

He resolves the conflict. When he's done, the result file overwrites the file in his branched client workspace; the file must then be submitted to the depot.

Integrating between unrelated files

If the target file was not branched from the source, there is no *base* (common ancestor) revision, and Perforce uses the first (most recently added) revision of the source file as its base revision. This operation is referred to as a *baseless merge*.

Integrating specific file revisions

By default, the `integrate` command integrates all the revisions following the last-integrated source revision into the target. To avoid having to manually delete unwanted revisions from the merge file while editing, you can specify a range of revisions to be integrated. If you are using `p4 integrate`, the ***`base`*** file is the closest common ancestor. If you are using `p4 merge`, the ***`base`*** file is the revision with the most edits in common.

Example 6.4. Integrating specific file revisions

Bruno has made two bug fixes to `//depot/dev/main/jam/scan.c` in the main codeline, and Earl wants to integrate the change into the release 1.0 branch. Although `scan.c` has gone through several revisions since the fixes were submitted, Earl knows that the bug fixes he wants were made to the 30th revision of `scan.c`. He types:

```
p4 integrate -b jamgraph-1.0-dev2release //depot/release/jam/1.0/scan.c#30,30
```

The target file (`//depot/release/jam/1.0/scan.c`) is given as an argument, but the file revisions are applied to the source. When Earl runs `p4 resolve`, only the the 30th revision of Bruno's file is scheduled for resolve. That is, Earl sees only the changes that Bruno made to `scan.c` at revision 30.

Reintegrating and reresolving files

After a revision of a source file has been integrated into a target, that revision is skipped in subsequent integrations to the same target. To force the integration of already-integrated files, specify the `-f` option when you issue the `p4 integrate` command.

A target that has been resolved but not submitted can be resolved again by specifying the `-f` option to `p4 resolve`. When you reresolve a file, ***yours*** is the new client file, the result of the original resolve.

Integration reporting

The following reporting commands provide useful information about the status of files being branched and integrated. Note the use of the preview option (`-n`) for reporting purposes.

To display this information	Use this command
Preview of the results of an integration	`p4 integrate -n [`*filepatterns*`]`
Files that are scheduled for resolve	`p4 resolve -n [`*filepatterns*`]`
Files that have been resolved but not yet submitted.	`p4 resolved`
List of branch specifications	`p4 branches`
The integration history of the specified files.	`p4 integrated` *filepatterns*
The revision histories of the specified files, including the integration histories of files from which the specified files were branched.	`p4 filelog -i [`*filepatterns*`]`

Streams

Perforce *streams* are "branches with brains," containers with which you compose a hierarchy that reflects your branching strategy. Streams are ideal for implementing the mainline branching model, in which volatile branches merge changes to keep up to date with their parents, then copy work to the parent when the work is stable enough to promote. Streams enable Perforce to generate views for associated workspaces, eliminating the need for you to update views manually to reflect changes to your branch structure.

For more information about streams, see the *P4V User's Guide*, "Working with Streams".

To work with streams, you perform the following steps:

1. Create a stream depot
2. Create and populate a mainline stream
3. Branch development and release streams
4. Merge and copy changes

To manage streams, you (mainly) use the following commands:

- `p4 stream`
- `p4 streams`
- `p4 merge`
- `p4 copy`
- `p4 resolve`
- `p4 cstat`
- `p4 istat`

Additional commands that accept stream arguments are as follows:

- `p4 branch`
- `p4 client`
- `p4 clients`
- `p4 diff2`
- `p4 dirs`
- `p4 integrate`
- `p4 interchanges`

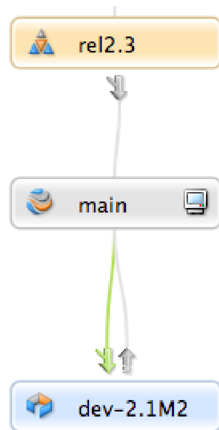
For details about command syntax and options, refer to the *P4 Command Reference* or issue the **`p4 help`** ***`commandname`*** command. For a brief overview, issue the **`p4 help streamintro`** command. The following sections describe streams-related tasks in detail.

Stream Types

You assign stream types according to the stream's expected usage, stability and flow of change. Stream types are as follows:

Stream Type	Stability	Expected Flow of Change	
		Merge	Copy
`mainline`	Stable per your policy (for example, all code builds)	from child (from release, or to development)	to child (to release, or from development)
`virtual`	N/A; used to filter streams	N/A	N/A
`development`	Unstable	from parent	to parent
`task`	Unstable	from parent	to parent
`release`	Highly stable	to parent	from parent

Merging means incorporating another stream's changes into your stream, and can require you to resolve conflicts. *Copy* propagates a duplicate of the source stream to the target. The following diagram shows a basic stream hierarchy: changes are merged down (to codelines of lesser stability) and copied up (to codelines of greater stability).



Stream types define default behavior for merging and copying, but you can override these defaults by editing the `toparent/notoparent` and `fromparent/nofromparent` options in the stream specification.

Stream Paths

Stream paths control the files and paths that compose a stream and define how those files are propagated. Except for the mainline, each stream inherits its structure from its parent stream. To modify the structure of the child, you specify paths as follows:

Type	Sync?	Submit?	Integrate to/ from Parent?	Remarks
`share`	Y	Y	Y	(Default) For files that are edited and propagated between parent and child streams. All files in a shared path are

Type	Sync?	Submit?	Integrate to/ from Parent?	Remarks
				branched and, in general, shared paths are the least restricted.
`isolate`	Y	Y	N	For files that must not be propagated outside the stream but can be edited within it, such as binary build results.
`import`	Y	N	N	For components that must be physically present in the stream but are never changed. Example: third-party libraries. Import paths can reference a specific changelist to limit the imported components to the revisions at that change or lower. Use the syntax ***`@changelist#`***, as in: `//depot/lib3.0/...@455678`
`import+`	Y	Y	N	Functions like an import path, in that it can reference an explicitly-defined depot path, but unlike a standard import path, you *can* submit changes to the files in an **`import+`** path.
`exclude`	N	N	N	Files in the parent stream that must never be part of the child stream.

In the following example, files in the **`src`** path are not submittable (and are imported from the parent stream's view), files in the **`lib`** path are not submittable (and are imported from an explicitly-specified location in the depot), and files in the **`db`** path can be edited and submitted in the stream, but can never be copied to the parent:

```
Paths:
    share ...
    import src/...
    import lib/...  //depot/lib3.0/...
    isolate db/...
```

The paths are used to generate the mappings for workspaces that are associated with the stream. If the stream structure changes, the clients views of associated workspaces are updated automatically and in fact cannot be altered manually. If the stream is **`locked`**, only the stream owner (or stream owners, if the **`Owner:`** field of the stream is set to a group) can edit the stream specification.

Stream specification can also remap file locations (so that a file in specified depot location is synced to a different location in the workspace) and screen out files according to file type. For example, to ensure that object files and executables are not part of the stream, add the following entries to the stream specification:

```
Ignored:
    .o
    .exe
```

Creating streams depots

Streams reside in *stream* (as opposed to *local*) depots. To create depots, you must have `super` privilege. To create a stream depot:

1. A Perforce superuser issues the`p4 depot` ***`depotname`*** command. The depot specification form is displayed. (Example: `p4 depot projectX`)
2. Set the `Type:` field to `stream`.
3. Adjust other settings as desired and save the specification.

Note that you cannot modify the type of a depot after you create it.

Creating streams

All streams reside one (path) level below the depot that contains them. For example, for Project X, expect a set of streams with depot paths like the following:

- `//projectX/main`
- `//projectX/dev-bruno`
- `//projectX/release1.0`
- `//projectX/release2.0`

...and so on.

First, create the mainline stream, which resides at the center of your stream hierarchy. Typically, the mainline is a fairly stable receiving trunk, accepting development work from child streams and propagating the results to release streams where it can be stabilized and built for release without impeding ongoing development.

To create a mainline:

1. Issue the `p4 stream` command, specifying the depot followed by the stream name. (Example: `p4 stream -t mainline //projectX/main`). The stream specification form is displayed.
2. Change options as required, and save the specification.
3. To verify that your mainline stream has been created, issue the `p4 streams` command. (Example: `p4 streams //projectX/...`).

Next, populate the mainline with files.

Populating the mainline

There are two ways to populate the mainline stream:

- Add files from the local filesystem, or
- Branch files from another depot.

If you need to preserve file history, branch the source files to the mainline stream. If you have no requirement for preserving file history, simply add them. The following sections describe each approach.

First, create a workspace

Before you can work in a stream, you must create a workspace associated with the stream. When you associate a workspace with a stream, Perforce generates the workspace view based on the structure of the stream. You never need to edit the workspace view (and, in fact, cannot manually alter it). If the structure of the stream changes, Perforce will update the views of workspaces associated with the stream on an as-needed basis.

(*Suggestion*: When assigning names to stream-associated workspaces, adopt a naming convention such as ***user_depot_streamname***. For example, **bruno_projectX**. If you regularly switch between client workspaces associated with different types of streams, you may also find it useful to append the stream type, to your workspace name, for example, **bruno_projectX_main** and **bruno_projectX_dev**.)

To create a workspace for a stream:

1. Issue the **p4 client** command, using the **-S** option to specify the name of the associated stream. (Example: **p4 client -S //projectX/main bruno_projectX**).
2. The workspace specification form is displayed. (Note the **Stream:** field, which is present only for stream-associated workspaces.)
3. Configure the workspace root directory and any other desired settings, and save the specification. You do not need to change the **View:** because this field will be maintained by Perforce.
4. To verify that your workspace has been created, issue the **p4 clients** command. For example: **p4 clients -S //projectX/main**.

Now you can populate the mainline with files.

Adding files

If you do not need to preserve the historic connection between the source files and the files in the new mainline stream, simply add them. To add files to the mainline stream:

1. Create the workspace root directory if it does not exist. For example:

```
cd C:\Users\bruno\p4clients\
mkdir bruno_projectX_main
```

2. Copy the files and folders to the workspace root directory.
3. Change into the client workspace root directory, and use the **p4 reconcile** command to detect files not under Perforce control and open them for add:

```
p4 reconcile -a
```

To verify that the files are set up to be added correctly, issue the `p4 opened` command. To populate the stream, submit the changelist in which the files are open.

Branching from other depots

You can branch files from other stream depots, classic depots, or remote depots. If you populate the mainline by branching, Perforce preserves the connection between the revision history of the source and target files. Your workspace must be set to one associated with the target stream (example: `p4 set P4CLIENT=bruno_projectX_main`).

To populate the mainline by branching, issue the `p4 copy` command, specifying source and target. Example:

```
p4 copy -v //mysourcedepot/mainline/... //ProjectX/main/...
```

(In this example, the `-v` option improves performance by updating the service without copying files to the workspace.) Perforce displays a series of "import from" messages listing the source and target files, and opens the file in a pending changelist. To preview the results of the operation without opening files, specify the `-n` option. To undo an erroneous copy operation, issue the p4 revert command; for example: `p4 revert //ProjectX/main/...`)

To verify that the files are set up to be added correctly, issue the `p4 opened` command. To populate the stream, `p4 submit` the changelist in which the files are open.

If you are populating an empty stream, you can simplify this process by using `p4 populate`. For example:

```
p4 populate //mysourcedepot/mainline/... //ProjectX/main/...
```

does the same thing as `p4 copy -v` followed by a `p4 submit`. If you are unsure of the results of `p4 populate`, use `p4 populate -n`, which previews the result of the command.

Populating child streams

After populating the mainline, you can branch files for development and for release. Development streams enable you to experiment without destabilizing the mainline, and release streams enable you to finalize existing features while working on new features in the mainline. For example, to create a development stream that is a clone of its mainline parent, issue the following command:

```
p4 stream -t development -P //projectX/main //projectX/dev
```

Perforce displays the stream specification with the type set to `development`. Save the specification. To populate the stream with the files from the mainline, issue the following commands:

```
p4 client -s -S //projectX/dev bruno_projectX_dev
p4 merge -S //projectX/dev -r
p4 submit -d "Branching from mainline"
```

Or just use `p4 populate`:

```
p4 populate -d "From main" -S //projectX/dev -P //projectX/main -r //projectX/dev
```

Propagating changes

Streams enable you to isolate stable code from work in progress, and to work concurrently on various projects without impediment. Best practice is to periodically update less stable streams from streams that are more stable (by merging), then promote changes to the more stable stream (by copying). Merging and copying are streamlined forms of integration. In general, propagate change as follows:

- For copying and branching, use `p4 copy`.
- For merging, use `p4 merge`.
- For edge cases not addressed by `p4 merge` or `p4 copy`, use `p4 integrate`.

The preceding guidelines apply both to streams and to classic depots.

Merging changes from a more-stable stream

To update a stream with changes from a more stable stream, issue the `p4 merge -S` *`source-stream`* command, resolve as required, and submit the resulting changelist. By default, you cannot copy changes to a more stable stream until you have merged any incoming changes from the intended target. This practice ensures that you do not inadvertently overwrite any of the contents of the more stable stream.

Assuming changes have been checked into the mainline after you started working in the development stream (and assuming your workspace is set to a development stream), you can incorporate the changes into the development stream by issuing the following commands:

```
p4 merge -S //projectX/dev -r
p4 resolve
p4 submit -d "Merged latest changes"
```

Copying changes to a more-stable stream

After merging, your stream is up to date with its more stable parent or child. Assuming you've finalized the changes you want to make in the development stream, you can now promote its new content with no danger of overwriting work in the target stream. The *copy* operation simply propagates a duplicate of the source to the target, with no resolve required. For example, (and assuming your workspace is set to a mainline parent stream) to promote changes from the development stream to its parent mainline, issue the following commands:

```
p4 copy -S //projectX/dev
p4 submit -d "Check my new feature in"
```

Propagating change across the stream hierarchy

You might need to propagate a specific change between two streams that do not have a natural parent-child relationship, for example, to obtain an in-progress feature or bug fix from a peer development stream. To merge from or copy to such a stream, you can reparent your stream by editing its specification and setting the `Parent` field to the desired source or target. This practice is not considered optimal but might be necessary.

Using task streams for sparse branching

Task streams are branches that work just like development streams, but they remain semi-private until branched back to the parent stream. Designed as lightweight branches, they are most effective when anticipated work in the branch will only affect a small number of files relative to the number of files in the branch.

Task streams do not require a parent stream. Because of this, even users who are not working in a stream depot can also take advantage of task streams. (The task stream must still reside in a stream depot, but your administrator can configure a stream depot as a dedicated holding place for task streams.)

Task streams are intended to be deleted or unloaded after use. Note that you cannot re-use task stream names even after the stream has been deleted; most sites will adopt a naming convention that is likely to be unique for each task, such as ***`user-date-jobnumber`***.

Working within task streams is just like working in a development stream:

1. Create the task stream (in this example, as a child of a development stream):

   ```
   p4 stream -t task -P //projectX/dev //Tasks/mybug123
   ```

2. Populate the stream:

   ```
   p4 populate -d "Fix bug 123" -S //Tasks/mybug123 -P //projectX/dev -r Tasks/mybug123
   ```

3. Make changes to files in the stream:

4. Merge down any required changes from the parent stream, resolving as necessary.

   ```
   p4 merge -S //Tasks/mybug123 -r
   ```

5. Copy up the changes you made into the parent stream:

   ```
   p4 copy -S //Tasks/mybug123 -P //projectX/dev
   ```

6. Delete or unload the task stream:

   ```
   p4 stream -d //Tasks/mybug123
   ```

   (Alternatively, use `p4 unload -s //Tasks/mybug123` to unload it.)

When in use, only the workspaces associated with the task stream can see all the files in the stream; the stream appears as a sparse branch to other workspaces, which see only those files and revisions that you changed within the task stream. Most other metadata for the task stream remains private.

Task streams can quickly accumulate in a depot until they are deleted or unloaded; to keep a project depot uncluttered by task streams, your Perforce administrator or project lead may choose to establish certain streams depots as dedicated holding areas for task streams. In this case, create your stream in the task streams depot as a child of a parent in the project depot.

Managing stream workspaces

This section discusses various approaches to managing your stream workspaces.

Using one workspace for multiple streams

Typically you define a workspace for each stream that you intend to work in. However, if your streams contain extremely large numbers of files (tens to hundreds of thousands, for example), this approach can require a time-consuming sync when you switch to working in a stream that has changed a lot since the last time you worked in it. If the structure of your streams is consistent and most files are identical, you can avoid this problem by reassociating your workspace with the stream you are switching to instead of using different workspaces for both streams. After you change the stream with which the workspace is associated, sync to obtain any files that differ.

To change the stream associated with a workspace, issue the following command:

`p4 client -s -S //`*`streamdepot`*`/ `*`streamname`*

Narrowing the scope of workspaces with virtual streams

For large projects, even consistently-organized streams may not sufficiently restrict workspace views. In large organizations, there are often many groups who are concerned with only a small subset of a project's files. In classic Perforce, these users would manually restrict their client workspace's view to include only the desired subset. Streams offers an alternative; use a **`virtual`** stream as a filter:

For example, if ongoing development work is occurring in an **`//Ace/dev`** stream:

```
Stream:  //Ace/dev
Parent: //Ace/main
Type:   development
Paths:
    share ...
```

Then a user who is working only with the documentation for the product (rather than all of the assets associated with the project) could create a virtual stream that includes only those files under **`//Ace/dev/docs/...`**, as follows:

```
Stream:  //Ace/devdocs
Parent: //Ace/dev
Type:   virtual
Paths:
    share docs/...
```

The user can then can switch his or her client workspace to the **`devdocs`** virtual stream with the following command:

```
p4 client -s -S //Ace/devdocs
```

When using the **devdocs** workspace, the user's client workspace view is automatically updated to include only the material in `//Ace/dev/docs/...` and any changes he or she makes in `//Ace/devdocs` are automatically propagated back to the original `//Ace/dev` codeline without the need to manually run `p4 copy` or `p4 integrate`.

Viewing a stream as of a specific changelist

The `StreamAtChange` option in the workspace specification lets you set a stream workspace to sync a stream using the stream's view as of a specific changelist. This gives you the ability to see a "back-in-time" view of your stream. This is helpful when you want to see what the stream view was at a particular point in time, especially if your stream definition changes a lot. When you use the `StreamAtChange` option, you cannot submit changes to the files in the stream, since your workspace files are not at the head revision.

To set a stream workspace to sync as of a specific changelist:

1. **Open the stream's workspace specification form for editing:**

   ```
   p4 client -S //projectX/main bruno_projectX
   ```

2. **Enter the changelist number in the `StreamAtChange` field in the `p4 client` form.**

For more information, see the *P4 Command Reference*, "p4 client".

Chapter 7

Labels

A Perforce *label* is a set of tagged file revisions. For example, you might want to tag the file revisions that compose a particular release with the label `release2.0.1`. In general, you can use labels to:

- Keep track of all the file revisions contained in a particular release of software.
- Distribute a particular set of file revisions to other users (for example, a standard configuration).
- Populate a clean build workspace.
- Specify a set of file revisions to be branched for development purposes.
- Sync the revisions as a group to a client workspace.

Labels and changelist numbers both refer to particular sets of file revisions but differ as follows:

- A label can refer to any set of file revisions. A changelist number refers to the contents of all the files in the depot at the time the changelist was submitted. If you need to refer to a group of file revisions from different points in time, use a label. If there is a point in time at which the files are consistent for your purposes, use a changelist number.
- You can change the contents of a label. You cannot change the contents of a submitted changelist.
- You can assign your own names to labels. Changelist numbers are assigned by Perforce.

Changelists are suitable for many applications that traditionally use labels. Unlike labels, changelists represent the state of a set of files at a specific time. Before you assume that a label is required, consider whether simply referring to a changelist number might fulfill your requirements.

Tagging files with a label

To tag a set of file revisions (in addition to any revisions that have already been tagged), use **`p4 tag`**, specifying a label name and the desired file revisions.

For example, to tag the head revisions of files that reside under **`//depot/release/jam/2.1/src/`** with the label **`jam-2.1.0`**, issue the following command:

```
p4 tag -l jam-2.1.0 //depot/release/jam/2.1/src/...
```

To tag revisions other than the head revision, specify a changelist number in the file pattern:

```
p4 tag -l jam-2.1.0 //depot/release/jam/2.1/src/...@1234
```

Only one revision of a given file can be tagged with a given label, but the same file revision can be tagged by multiple labels.

Untagging files

You can untag revisions with:

```
p4 tag -d -l labelname filepattern
```

This command removes the association between the specified label and the file revisions tagged by it. For example, if you have tagged all revisions under `//depot/release/jam/2.1/src/...` with `jam-2.1.0`, you can untag only the header files with:

```
p4 tag -d -l jam-2.1.0 //depot/release/jam/2.1/src/*.h
```

Previewing tagging results

You can preview the results of `p4 tag` with `p4 tag -n`. This command lists the revisions that would be tagged, untagged, or retagged without actually performing the operation.

Listing files tagged by a label

To list the revisions tagged with *`labelname`*, use `p4 files`, specifying the label name as follows:

```
p4 files @labelname
```

All revisions tagged with *`labelname`* are listed, with their file type, change action, and changelist number. (This command is equivalent to `p4 files //...@labelname`).

Listing labels that have been applied to files

To list all labels that have been applied to files, use the command:

```
p4 labels filepattern
```

Using a label to specify file revisions

You can use a label name anywhere you can refer to files by revision (`#1`, `#head`), changelist number (`@7381`), or date (`@2011/07/01`).

If you omit file arguments when you issue the `p4 sync @labelname` command, all files in the client workspace view that are tagged by the label are synced to the revision specified in the label. All files in the workspace that do not have revisions tagged by the label are deleted from the workspace. Open files or files not under Perforce control are unaffected. This command is equivalent to `p4 sync //...@labelname`.

If you specify file arguments when you issue the `p4 sync` command (`p4 sync files@labelname`), files that are in your workspace and tagged by the label are synced to the tagged revision.

Example 7.1. Retrieving files tagged by a label into a client workspace

To retrieve the files tagged by Earl's `jam-2.1.0` label into his client workspace, Bruno issues the following command:

```
p4 sync @ jam-2.1.0
```

and sees:

```
//depot/dev/main/jam/Build.com#5 - updating c:\bruno_ws\dev\main\jam\Build.com
//depot/dev/main/jam/command.c#5 - updating c:\bruno_ws\dev\main\jam\command.c
//depot/dev/main/jam/command.h#3 - added as c:\bruno_ws\dev\main\jam\command.h
//depot/dev/main/jam/compile.c#12 - updating c:\bruno_ws\dev\main\jam\compile.c
//depot/dev/main/jam/compile.h#2 - updating c:\bruno_ws\dev\main\jam\compile.h
<etc>
```

Deleting labels

To delete a label, use the following command:

`p4 label -d` *`labelname`*

Deleting a label has no effect on the tagged file revisions (though, of course, the revisions are no longer tagged).

Creating a label for future use

To create a label without tagging any file revisions, issue the **`p4 label`** ***`labelname`*** command. This command displays a form in which you describe and specify the label. After you have created a label, you can use **`p4 tag`** or **`p4 labelsync`** to apply the label to file revisions.

Label names cannot be the same as client workspace, branch, or depot names.

For example, to create **`jam-2.1.0,`** issue the following command:

`p4 label jam-2.1.0`

The following form is displayed:

```
Label:    jam-2.1.0
Update:    2011/03/07 13:07:39
Access:    2011/03/07 13:13:35
Owner:    earl
Description:
     Created by earl.
Options:     unlocked noautoreload
View:
     //depot/...
```

Enter a description for the label and save the form. (You do not need to change the **`View:`** field.)

After you create the label, you are able to use the **`p4 tag`** and **`p4 labelsync`** commands to apply the label to file revisions.

Restricting files that can be tagged

The **`View:`** field in the **`p4 label`** form limits the files that can be tagged with a label. The default label view includes the entire depot (**`//depot/...`**). To prevent yourself from inadvertently tagging every file in your depot, set the label's **View:** field to the files and directories to be taggable, using depot syntax.

Example 7.2. Using a label view to control which files can be tagged

Earl wants to tag the revisions of source code in the release 2.1 branch, which he knows can be successfully compiled. He types `p4 label jam-2.1.0` and uses the label's **View:** field to restrict the scope of the label as follows:

```
Label:    jam-2.1.0
Update:    2011/03/07 13:07:39
Access:    2011/03/07 13:13:35
Owner:    earl
Description:
     Created by earl.
Options:     unlocked noautoreload
View:
     //depot/release/jam/2.1/src/...
```

This label can tag only files in the release 2.1 source code directory.

Using static labels to archive workspace configurations

You can use static labels to archive the state of your client workspace (meaning the currently synced file revisions) by issuing the `p4 labelsync` command. The label you specify must have the same view as your client workspace.

For example, to record the configuration of your current client workspace using the existing `ws_config` label, use the following command:

```
p4 labelsync -l ws_config
```

All file revisions that are synced to your current workspace and visible through both the client workspace view and the label view (if any) are tagged with the `ws_config` label. Files that were previously tagged with `ws_config`, then subsequently removed from your workspace (`p4 sync #none`), are untagged.

To sync the files tagged by the `ws_config` label (thereby recreating the workspace configuration), issue the following command:

```
p4 sync @ws_config
```

Note

You can control how static labels are stored using the `autoreload` or `noautoreload` option:

- `autoreload` stores the labels in the unload depot. This storage option can improve performance on sites that make heavy use of labels.
- `noautoreload` stores the labels in the `db.label` table.

> These storage options do not affect automatic labels.

Using automatic labels as aliases for changelists or other revisions

You can use automatic labels to specify files at certain revisions without having to issue the **p4 labelsync** command.

To create an automatic label, fill in the **Revision:** field of the **p4 label** form with a revision specifier. When you sync a workspace to an automatic label, the contents of the **Revision:** field are applied to every file in the **View:** field.

Example 7.3. Using an automatic label as an alias for a changelist number.

Earl is running a nightly build process, and has successfully built a product as of changelist 1234. Rather than having to remember the specific changelist for every night's build, he types **p4 label nightly20111201** and uses the label's **Revision:** field to automatically tag all files as of changelist 1234 with the **nightly20111201** label:

```
Label:    nightly20111201
Owner:    earl
Description:
     Nightly build process.
Options:     unlocked noautoreload
View:
     //depot/...
Revision:
     @1234
```

The advantage to this approach is that it is highly amenable to scripting, takes up very little space in the label table, and provides a way to easily refer to a nightly build without remembering which changelist number was associated with the night's build process.

Example 7.4. Referring specifically to the set of files submitted in a single changelist.

A bug was fixed by means of changelist 1238, and requires a patch label that refers to only those files associated with the fix. Earl types **p4 label patch20111201** and uses the label's **Revision:** field to automatically tag only those files submitted in changelist 1238 with the **patch20111201** label:

```
Label:    patch20111201
Owner:    earl
Description:
     Patch to 2011/12/01 nightly build.
Options:     unlocked noautoreload
View:
     //depot/...
Revision:
     @1238,1238
```

This automatic label refers only to those files submitted in changelist 1238.

Example 7.5. Referring to the first revision of every file over multiple changelists.

You can use revision specifiers other than changelist specifiers; in this example, Earl is referring to the first revision (#1) of every file in a branch. Depending on how the branch was populated, these files could have been created through multiple changelists over a long period of time:

```
Label:    first2.2
Owner:    earl
Description:
     The first revision in the 2.2 branch
Options:     unlocked noautoreload
View:
     //depot/release/jam/2.2/src/...
Revision:
     "#1"
```

Because Perforce forms use the `#` character as a comment indicator, Earl has placed quotation marks around the `#` to ensure that it is parsed as a revision specifier.

Preventing inadvertent tagging and untagging of files

To tag the files that are in your client workspace and label view (if set) and untag all other files, issue the `p4 labelsync` command with no arguments. To prevent the inadvertent tagging and untagging of files, issue the `p4 label` ***`labelname`*** command and lock the label by setting the **`Options:`** field of the label form to `locked`. To prevent other users from unlocking the label, set the **`Owner:`** field. For details about Perforce privileges, refer to the *Perforce Server Administrator's Guide: Fundamentals*.

Using labels on edge servers

You can user Perforce Server in a *distributed*, multi-site environment using central and edge servers. With a distributed Perforce service architecture, users typically connect to an edge server and execute commands just as they would with a *classic* Perforce service. For more information, refer to *Perforce Server Administrator's Guide: Multi-site Deployment*.

When connected to an edge server, the commands `p4 label`, `p4 labelsync`, and `p4 tag` operate on labels local to the edge server. Global labels can be manipulated by using the `-g` option. For details, refer to the *P4 Command Reference*.

Note Using the `-g` option with `p4 labelsync` only works with a global client. To manipulate a global label, use `p4 tag`.

Chapter 8 Defect Tracking

A *job* is a numbered (or named) work request managed by Perforce. Perforce jobs enable you to track the status of bugs and enhancement requests and associate them with changelists that implement fixes and enhancements. You can search for jobs based on the contents of fields, the date the job was entered or last modified, and many other criteria.

Your Perforce administrator can customize the job specification for your site's requirements. For details on modifying the job specification, see the *Perforce Server Administrator's Guide: Fundamentals*.

To integrate Perforce with your in-house defect tracking system, or to develop an integration with a third-party defect tracking system, use P4DTG, the Perforce Defect Tracking Gateway. P4DTG is an integrated platform that includes both a graphical configuration editor and a replication engine. For more information, see:

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

Managing jobs

To create a job using Perforce's default job-naming scheme, issue the `p4 job` command. To assign a name to a new job (or edit an existing job), issue the `p4 job jobname` command.

Example 8.1. Creating a job

Gale discovers about a problem with Jam, so she creates a job by issuing the `p4 job` command and describes it as follows:

```
Job:     job000006
Status:  open
User:    gale
Date:    2011/11/14 17:12:21
Description:
    MAXLINE can't account for expanded cmd buffer size.
```

The following table describes the fields in the default job specification.

Field Name	Description	Default
`Job`	The name of the job (white space is not allowed). By default, Perforce assigns job names using a numbering scheme (`jobnnnnnn`).	Last job number + 1
`Status`	• `open:` job has not yet been fixed. • `closed:` job has been completed. • `suspended:` job is not currently being worked on.	`open`
`User`	The user to whom the job is assigned, usually the person assigned to fix this particular problem.	Perforce user name of the job creator.

Field Name	Description	Default
`Date`	The date the job was last modified.	Updated by Perforce when you save the job.
`Description`	Describes the work being requested, for example a bug description or request for enhancement.	None. You must enter a description.

To edit existing jobs, specify the job name when you issue the `p4 job` command: `p4 job` ***`jobname`***. Enter your changes in the job form, save the form and exit.

To delete a job, issue the `p4 job -d` ***`jobname`*** command.

Searching jobs

To search Perforce jobs, issue the `p4 jobs -e` ***`jobview`*** command, where ***`jobview`*** specifies search expressions described in the following sections. For more details, issue the `p4 help jobview` command.

Searching job text

You can use the expression `'`***`word1 word2 ... wordN`***`'` to find jobs that contain all of ***`word1`*** through ***`wordN`*** in any field (excluding date fields). Use single quotes on UNIX and double quotes on Windows.

When searching jobs, note the following restrictions:

- When you specify multiple words separated by whitespace, Perforce searches for jobs that contain *all* the words specified. To find jobs that contain *any* of the terms, separate the terms with the pipe (`|`) character.
- Field names and text comparisons in expressions are not case-sensitive.
- Only alphanumeric text and punctuation can appear in an expression. To match the following characters, which are used by Perforce as logical operators, precede them with a backslash: `=^&|()<>`.
- You cannot search for phrases, only individual words.

Example 8.2. Searching jobs for specific words

Bruno wants to find all jobs that contain the words `filter`, `file`, and `mailbox`. He types:

```
p4 jobs -e 'filter file mailbox'
```

Example 8.3. Finding jobs that contain any of a set of words in any field

Bruno wants to find jobs that contain any of the words `filter`, `file` or `mailbox`. He types:

```
p4 jobs -e 'filter|file|mailbox'
```

You can use the `*` wildcard to match one or more characters. For example, the expression `fieldname=string*` matches `string`, `strings`, `stringbuffer`, and so on.

To search for words that contain wildcards, precede the wildcard with a backslash in the command. For instance, to search for `*string` (perhaps in reference to `char *string`), issue the following command:

```
p4 jobs -e '\*string'
```

Searching specific fields

To search based on the values in a specific field, specify ***`field=value`***.

Example 8.4. Finding jobs that contain words in specific fields

Bruno wants to find all open jobs related to filtering. He types:

```
p4 jobs -e 'Status=open User=bruno filter.c'
```

This command finds all jobs with a `Status:` of `open`, a `User:` of `bruno`, and the word `filter.c` in any nondate field.

To find fields that do not contain a specified expression, precede it with `^`, which is the NOT operator. The NOT operator `^` can be used only directly after an AND expression (space or `&`). For example, `p4 jobs -e '^user=bruno'` is not valid. To get around this restriction, use the `*` wildcard to add a search term before the ^ term; for example: `p4 jobs -e 'job=* ^user=bruno'` returns all jobs not owned by Bruno.

Example 8.5. Excluding jobs that contain specified values in a field

Bruno wants to find all open jobs he does not own that involve filtering. He types:

```
p4 jobs -e 'status=open ^user=bruno filter'
```

This command displays all open jobs that Bruno does not own that contain the word `filter`.

Using comparison operators

The following comparison operators are available: `=`, `>`, `<`, `>=`, `<=`, and `^` for Boolean NOT.

The behavior of these operators depends upon the type of the field in the expression. The following table describes the field types and how they can be searched.

Field Type	Description	Notes
`word`	A single word	The equality operator (`=`) matches the value in the word field exactly. The relational operators perform comparisons in ASCII order.
`text`	A block of text entered on the lines beneath the field name.	The equality operator (`=`) matches the job if the value is found anywhere in the specified field.

Field Type	Description	Notes
		The relational operators are of limited use here, because they'll match the job if *any* word in the specified field matches the provided value. For example, if a job has a text field **ShortDescription:** that contains only the phrase **gui bug**, and the expression is **'ShortDesc<filter'**, the job will match the expression, because **bug<filter**.
line	A single line of text entered on the same line as the field name.	Same as **text**
select	One of a set of values. For example, job status can be **open**/**suspended**/**closed**.	The equality operator (=) matches a job if the value in the field is the specified word. Relational operators perform comparisons in ASCII order.
date	A date and optionally a time. For example, **2011/07/15:13:21:40**	Dates are matched chronologically. If a time is not specified, the operators =, <=, and >= match the whole day.
bulk	Like **text**, but not indexed for searching.	These fields are not searchable with **p4 jobs -e**.

If you're not sure of a field's type, issue the **p4 jobspec -o command**, which displays your job specification. The field called **Fields:** lists the job fields' names and data types.

Searching date fields

To search date fields, specify the date using the format ***yyyy/mm/dd*** or ***yyyy/mm/dd:hh:mm:ss***. If you omit time, the equality operator (=) matches the entire day.

Example 8.6. Using dates within expressions

Bruno wants to view all jobs modified on July 13, 2011. He enters:

```
p4 jobs -e 'ModifiedDate=2011/07/13'
```

Fixing jobs

To fix a job, you link it to a changelist and submit the changelist. Perforce automatically changes the value of a job's status field to **closed** when the changelist is submitted.

Jobs can be linked to changelists in one of three ways:

- By setting the **JobView:** field in the **p4 user** form to an expression that matches the job.

- With the `p4 fix` command.
- By editing the `p4 submit` form.

You can modify job status directly by editing the job, but if you close a job manually, there's no association with the changelist that fixed the job. If you have altered your site's job specification by deleting the `Status:` field, jobs can still be linked to changelists, but status cannot be changed when the changelist is submitted. (In most cases, this is not a desired form of operation.) See the chapter on editing job specifications in the *Perforce Server Administrator's Guide: Fundamentals* for more details.

To remove jobs from a changelist, issue the `p4 fix -d` command.

Linking automatically

You can modify your Perforce user specification to automatically attach open jobs to any changelists you create. To set up automatic inclusion, issue the `p4 user` command and set the `JobView:` field value to a valid expression that locates the jobs you want attached.

Example 8.7. Automatically linking jobs to changelists

Bruno wants to see all open jobs that he owns in all changelists he creates. He types `p4 user` and adds the `JobView:` field:

```
User:     bruno
Update:   2011/06/02 13:11:57
Access:   2011/06/03 20:11:07
JobView:  user=bruno&status=open
```

All of Bruno's open jobs now are automatically attached to his default changelist. When he submits changelists, he must be sure to delete jobs that aren't fixed by the changelist he is submitting.

Linking manually

To link a job to a changelist manually, issue the `p4 fix -c` *`changenum jobname`* command. If the changelist has already been submitted, the value of the job's `Status:` field is changed to `closed`. Otherwise, the status is not changed.

Example 8.8. Manually linking jobs to changelists

You can use `p4 fix` to link a changelist to a job owned by another user.

Sarah has just submitted a job called `options-bug` to Bruno, but the bug has already been fixed in Bruno's previously submitted changelist 18. Bruno links the job to the changelist by typing:

```
p4 fix -c 18 options-bug
```

Because changelist 18 has already been submitted, the job's status is changed to `closed`.

Linking jobs to changelists

To link jobs to changelists when submitting or editing the changelist, enter the job names in the `Jobs:` field of the changelist specification. When you submit the changelist, the job is (by default) closed.

To unlink a job from a pending changelist, edit the changelist and delete its name from the `Jobs:` field. To unlink a job from a submitted changelist, issue the `p4 fix -d -c` *`changenum jobname`* command.

Chapter 9

Scripting and Reporting

This chapter provides details about using **p4** commands in scripts and for reporting purposes. For a full description of any particular command, consult the *P4 Command Reference*, or issue the **p4 help** command.

Common options used in scripting and reporting

The following command-line options enable you to specify settings on the command line and in scripts. For full details, refer to the description of global options in the *P4 Command Reference*.

Option	Description
`-b` *batchsize*	Specify a batch size (number of arguments) to use when processing a command from `-x` *argfile*. By default, 128 arguments are read at a time.
`-c` *client_workspace*	Specifies the client workspace name.
`-G`	Causes all output (and batch input for form commands with `-i`) to be formatted as marshaled Python dictionary objects.
`-p` *protocol*:*host*:*port*	Specifies the host and port number of the Perforce service, as well as the protocol used to connect.
`-P` *password*	Specifies the user password if any. If you prefer your script to log in before running commands (instead of specifying the password every time a command is issued), use the `p4 login` command. For example: `echo 'mypassword' | p4 login`
`-s`	Prepends a descriptive field (for example, `text:`, `info:`, `error:`, `exit:`) to each line of output produced by a Perforce command.
`-u` *user*	Specifies the Perforce user name.
`-x` *argfile*	Reads arguments, one per line, from the specified file. If *argfile* is a single hyphen (`-`), then standard input is read.

Scripting with Perforce forms

If your scripts issue **p4** commands that require the user to fill in a form, such as the **p4 client** and **p4 submit** commands, use the the **-o** option to write the form to standard output and the **-i** option to read the edited form from standard input.

For example, to create a job using a script on UNIX:

1. Issue the **p4 job -o > temp1** command to write a blank job specification into a text file.

2. Make the necessary changes to the job. For example:

```
sed 's/<enter description here>/Crashes on exit./' temp1 > temp2
```

3. Issue the `p4 job -i < temp2` command to save the job.

To accomplish the preceding without a temporary file, issue the following command:

```
p4 job -o | sed 's/<enter description here>/Crashes on exit./' | p4 job -i
```

The commands that display forms are:

- `p4 branch`
- `p4 change`
- `p4 client`
- `p4 job`
- `p4 label`
- `p4 submit` (use`p4 change -o` to create changelist, or `p4 submit -d "A changelist description"` to supply a description to the default changelist during changelist submission.)
- `p4 stream`
- `p4 user`

File reporting

The following sections describe commands that provide information about file status and location. The following table lists a few basic and highly-useful reporting commands.

To display this information	Use this command
File status, including file type, latest revision number, and other information	`p4 files`
File revisions from most recent to earliest	`p4 filelog`
Currently opened files	`p4 opened`
Preview of `p4 sync` results	`p4 sync -n`
Summarize a `p4 sync` preview, estimate network traffic	`p4 sync -N`
Currently synced files	`p4 have`
The contents of specified files	`p4 print`
The mapping of files' depot locations to the corresponding workspace locations.	`p4 where`

To display this information	Use this command
A list of files and full details about the files	`p4 fstat`

Displaying file status

To display information about single revisions of files, issue the `p4 files` command. This command displays the locations of the files in the depot, the actions (`add`, `edit`, `delete`, and so on) performed on those files at the specified revisions, the changelists in which the specified file revisions were submitted, and the files' types. The following example shows typical output of the `p4 files` command:

```
//depot/README#5 - edit change 6 (text)
```

The `p4 files` command requires one or more *`filespec`* arguments. Regardless of whether you use local, client, or depot syntax to specify the filespec arguments, the `p4 file` command displays results using depot syntax. If you omit the revision number, information for the head revision is displayed. The output of `p4 files` includes deleted revisions.

The following table lists some common uses of the `p4 files` command.

To display the status of	Use this command
All files in the depot, regardless of your client workspace view For depots containing numerous files, you can maximize performance by avoiding commands that refer to the entire depot (`//depot/...`) when not required. For best performance, specify only the directories and files of interest.	`p4 files //depot/...`
The files currently synced to the specified client workspace	`p4 files @`*`workspacename`*
The files mapped by your client workspace view	`p4 files //`*`workspacename`*`/...`
Specified files in the current working directory	`p4 files` *`filespec`*
A specified file revision	`p4 files` *`filespec#rev`*
Specified files at the time a changelist was submitted, regardless of whether the files were submitted in the changelist	`p4 files` *`filespec@changenum`*
Files tagged with a specified label	`p4 files` *`filespec@labelname`*

Displaying file revision history

To display the revision history of a file, issue the `p4 filelog` *`filespec`* command. The following example shows how `p4 filelog` displays revision history.

```
p4 filelog //depot/dev/main/jam/jam.c

//depot/dev/main/jam/jam.c
... #35 change 627 edit on 2011/11/13 by earl@earl-dev-yew (text)
'Handle platform variants better'
... #34 change 598 edit on 2011/10/24 by raj@raj-althea (text)
'Reverse previous attempt at fix'
... ... branch into //depot/release/jam/2.2/src/jam.c#1
... #33 change 581 edit on 2011/10/03 by gale@gale-jam-oak (text)
'Version strings & release notes'
```

To display the entire description of each changelist, specify the `-l` option.

Listing open files

To list the files that are currently opened in a client workspace, issue the `p4 opened` *`filespec`* command. The following line is an example of the output displayed by the `p4 opened` command:

```
//depot/dev/main/jam/fileos2.c- edit default change (text)
```

The following table lists some common uses of the `p4 opened` command.

To list	Use this command
Opened files in the current workspace	`p4 opened`
Opened files in all client workspaces	`p4 opened -asp4 opened -a`
Files in a numbered pending changelist	`p4 opened -c changelist`
Files in the default changelist	`p4 opened -c default`
Whether a specific file is opened by you	`p4 opened filespec`
Whether a specific file is opened by anyone	`p4 opened -a filespec`

Displaying file locations

To display information about the locations of files, use the`p4 where`, `p4 have`, and `p4 sync -n` commands:

- To display the location of a file in depot, client, and local syntax, issue the `p4 where` command.
- To list the location and revisions of files that you last synced to your client workspace, issue the `p4 have` command.
- To see where files will be synced in your workspace, preview the sync by issuing the `p4 sync -n` command.

You can use these commands with or without *filespec* arguments.

The following table lists some useful location reporting commands.

To display	Use this command
The revision number of a file that you synced to your workspace	`p4 have` *`filespec`*
How a particular file in the depot maps to your workspace	`p4 where //depot/`*`filespec`*

Displaying file contents

To display the contents of a file in the depot, issue the `p4 print` *`filespec`* command. This command prints the contents of the file to standard output or to a specified output file, with a one-line banner that describes the file. To suppress the banner, specify the `-q` option. By default, the head revision is displayed, but you can specify a file revision.

To display the contents of files	Use this command
At the head revision	`p4 print` *`filespec`*
Without the banner	`p4 print -q` *`filespec`*
At a specified changelist number	`p4 print` *`filespec@changenum`*

Displaying annotations (details about changes to file contents)

To find out which file revisions or changelists affected lines in a text file, issue the `p4 annotate` command.

By default, `p4 annotate` displays the file line by line, with each line preceded by a revision number indicating the revision that made the change. To display changelist numbers instead of revision numbers, specify the `-c` option.

Example 9.1. Using p4 annotate to display changes to a file.

A file is added (`file.txt#1`) to the depot, containing the following lines:

```
This is a text file.
The second line has not been changed.
The third line has not been changed.
```

The third line is deleted and the second line edited so that `file.txt#2` reads:

```
This is a text file.
The second line is new.
```

The output of `p4 annotate` and `p4 annotate -c` look like this:

```
$ p4 annotate file.txt

//depot/files/file.txt#3 - edit change 153 (text)
1: This is a text file.
2: The second line is new.

$ p4 annotate -c file.txt

//depot/files/file.txt#3 - edit change 153 (text)
151: This is a text file.
152: The second line is new.
```

The first line of `file.txt` has been present since revision 1, which was submitted in changelist 151. The second line has been present since revision 2, which was submitted in changelist 152.

To show all lines (including deleted lines) in the file, use `p4 annotate -a` as follows:

```
$ p4 annotate -a file.txt

//depot/files/file.txt#3 - edit change 12345 (text)
1-3: This is a text file.
1-1: The second line has not been changed.
1-1: The third line has not been changed.
2-3: The second line is new.
```

The first line of output shows that the first line of the file has been present for revisions 1 through 3. The next two lines of output show lines of `file.txt` present only in revision 1. The last line of output shows that the line added in revision 2 is still present in revision 3.

You can combine the `-a` and `-c` options to display all lines in the file and the changelist numbers (rather than the revision numbers) at which the lines existed.

Monitoring changes to files

To track changes to files as they occur, you can use the Perforce change review daemon, which enables Perforce users to specify files and directories of interest and receive email when a changelist that affects the specified files is submitted. For details about administering the review daemon, refer to the *Perforce Server Administrator's Guide: Fundamentals* and to the description of the `p4 review` command in the *P4 Command Reference*.

The following table lists commands that display information about the status of files, changelists, and users. These commands are often used in review daemons.

To list	Use this command
The users who review specified files	`p4 reviews` *`filespec`*
The users who review files in a specified changelist	`p4 reviews -c` *`changenum`*
A specified user's email address	`p4 users` *`username`*

Changelist reporting

The `p4 changes` command lists changelists that meet search criteria, and the `p4 describe` command lists the files and jobs associated with a specified changelist. These commands are described below.

Listing changelists

To list changelists, issue the `p4 changes` command. By default, `p4 changes` displays one line for every public changelist known to the system, as well as for any restricted changelists to which you have access. The following table lists command-line options that you can use to filter the list.

To list changelists	Use this command
With the first 31 characters of the changelist descriptions	`p4 changes`
With full descriptions	`p4 changes -l`
The last *n* changelists	`p4 changes -m` *`n`*

With a specified status	`p4 changes -s pending`
	`p4 changes -s submitted`
	`p4 changes -s shelved`

From a specified user	`p4 changes -u` *`user`*
From a specified workspace	`p4 changes -c` *`workspace`*
That affect specified files	`p4 changes` *`filespec`*
That affect specified files, including changelists that affect files that were later integrated with the named files	`p4 changes -i` *`filespec`*
That affect specified files, including only those changelists between revisions *m* and *n* of these files	`p4 changes` *`filespec#m,#n`*
That affect specified files at each revision between the revisions specified in labels *`lab1`* and *`lab2`*	`p4 changes` *`filespec@lab1,@lab2`*
Submitted between two dates	`p4 changes` *`@date1,@date2`*
Submitted on or after a specified date	`p4 changes` *`@date1,@now`*

Listing files and jobs affected by changelists

To list files and jobs affected by a specified changelist, along with the diffs of the changes, issue the **p4 describe** command. To suppress display of the diffs (for shorter output), specify the **-s** option. The following table lists some useful changelist reporting commands.

To list	Use this command
Files in a pending changelist	`p4 opened -c` *changenum*
Files submitted and jobs fixed by a particular changelist, including diffs	`p4 describe` *changenum*
Files submitted and jobs fixed by a particular changelist, suppressing diffs	`p4 describe -s` *changenum*
Files and jobs affected by a particular changelist, passing the context diff option to the underlying diff program	`p4 describe -dc` *changenum*
The state of particular files at a particular changelist, regardless of whether these files were affected by the changelist	`p4 files` *filespec@changenum*

For more commands that report on jobs, see "Job reporting" on page 99.

Label reporting

To display information about labels, issue the **p4 labels** command. The following table lists some useful label reporting commands.

To list	Use this command
All labels, with creation date and owner	`p4 labels`
All labels containing a specific file revision (or range)	`p4 labels file#` *revrange*
Files tagged with a specified label	`p4 files` *@labelname*
A preview of the results of syncing to a label	`p4 sync -n` *@labelname*

Branch and integration reporting

The following table lists commonly used commands for branch and integration reporting.

To list	Use this command
All branch specifications	`p4 branches`
Files in a specified branch	`p4 files` *filespec*

To list	Use this command
The revisions of a specified file	`p4 filelog` *filespec*
The revisions of a specified file, recursively including revisions of the files from which it was branched	`p4 filelog -i` *filespec*
A preview of the results of a resolve	`p4 resolve [`*args*`] -n [`*filespec*`]`
Files that have been resolved but not yet submitted	`p4 resolved [`*filespec*`]`
Integrated, submitted files that match the *filespec* arguments	`p4 integrated` *filespec*
A preview of the results of an integration	`p4 integrate [`*args*`] -n [`*filespec*`]`

Job reporting

Listing jobs

To list jobs, issue the `p4 jobs` command. The following table lists common job reporting commands.

To list	Use this command
All jobs	`p4 jobs`
All jobs, including full descriptions	`p4 jobs -l`
Jobs that meet search criteria (see "Searching jobs" on page 86 for details)	`p4 jobs -e` *jobview*
Jobs that were fixed by changelists that contain specific files	`p4 jobs` *filespec*
Jobs that were fixed by changelists that contain specific files, including changelists that contain files that were later integrated into the specified files	`p4 jobs -i` *filespec*

Listing jobs fixed by changelists

Any jobs that have been linked to a changelist with `p4 change`, `p4 submit`, or `p4 fix` are referred to as *fixed* (regardless of whether their status is `closed`). To list jobs that were fixed by changelists, issue the `p4 fixes` command.

The following table lists useful commands for reporting fixes.

To list	Use this command
all changelists linked to jobs	`p4 fixes`

To list	Use this command
all changelists linked to a specified job	`p4 fixes -j` *jobname*
all jobs linked to a specified changelist	`p4 fixes -c` *changenum*
all fixes associated with specified files	`p4 fixes` *filespec*
all fixes associated with specified files, including changelists that contain files that were later integrated with the specified files	`p4 fixes -i` *filespec*

System configuration reporting

The commands described in this section display Perforce users, client workspaces, and depots.

Displaying users

The `p4 users command` displays the user name, an email address, the user's "real" name, and the date that Perforce was last accessed by that user, in the following format:

```
bruno <bruno@bruno_ws> (bruno) accessed 2011/03/07
dai <dai@dai_ws> (Dai Sato) accessed 2011/03/04
earl <earl@earl_ws> (Earl Ashby) accessed 2011/03/07
gale <gale@gale_ws> (Gale Beal) accessed 2011/06/03
hera <hera@hera_ws> (Hera Otis) accessed 2011/10/03
ines <ines@ines_ws> (Ines Rios) accessed 2011/02/02
jack <jack@submariner> (jack) accessed 2011/03/02
mei <mei@mei_ws> (Mei Chang) accessed 2011/11/14
ona <ona@ona_ws> (Ona Birch) accessed 2011/10/23
quinn <quinn@quinn_ws> (Quinn Cass) accessed 2011/01/27
raj <raj@ran_ws> (Raj Bai) accessed 2011/07/28
vera <vera@vera_ws> (Vera Cullen) accessed 2011/01/15
```

Displaying workspaces

To display information about client workspaces, issue the `p4 clients` command, which displays the client workspace name, the date the workspace was last updated, the workspace root, and the description of the workspace, in the following format.

```
Client bruno_ws 2011/03/07 root c:\bruno_ws ''
Client earl-dev-beech 2011/10/26 root /home/earl ''
Client earl-dev-guava 2011/09/08 root /usr/earl/development ''
Client earl-dev-yew 2011/11/19 root /tmp ''
Client earl-win-buckeye 2011/03/21 root c:\src ''
Client earl-qnx-elm 2011/01/17 root /src ''
Client earl-tupelo 2011/01/05 root /usr/earl ''
```

Listing depots

To list depots, issue the **p4 depots** command. This command lists the depot's name, its creation date, its type (`local`, `remote`, `archive`, `spec`, or `stream`), its host name or IP address (if `remote`), the mapping to the local depot, and the system administrator's description of the depot.

For details about defining multiple depots on a single Perforce installation, see the *Perforce Server Administrator's Guide: Fundamentals*.

Sample script

The following sample script parses the output of the **p4 fstat** command to report files that are opened where the head revision is not in the client workspace (a potential problem).

Example 9.2. Sample shell script showing parsing of p4 fstat command output.

```
#!/bin/sh
# Usage: opened-not-head.sh files
# Displays files that are open when the head revision is not
# on the client workspace

echo=echo
exit=exit
p4=p4
sed=sed

if [ $# -ne 1 ]
then
     $echo "Usage: $0 files"
     $exit 1
fi

$p4 fstat -Ro $1 | while read line
do
     name=`$echo $line | $sed 's/^[\. ]\+\([^ ]\+\) .*$/\1/'`
     value=`$echo $line | $sed 's/^[\. ]\+[^ ]\+ \(.*\)$/\1/'`

     if [ "$name" = "depotFile" ]
     then
         depotFile=$value

     elif [ "$name" = "headRev" ]
     then
         headRev=$value

     elif [ "$name" = "haveRev" ]
     then
         haveRev=$value

         if [ $headRev != $haveRev ]
         then
             $echo $depotFile
         fi
     fi
done
```

Appendix

Glossary

Term	Definition
access level	A permission assigned to a user to control which Perforce commands the user can execute. See *protections*.
admin access	An access level that gives the user permission to run Perforce commands that override *metadata* but do not affect the state of the service.
apple file type	Perforce file type assigned to files that are stored using AppleSingle format, permitting the data fork and resource fork to be stored as a single file.
atomic change transaction	Grouping operations affecting a number of files in a single transaction. If all operations in the transaction succeed, all the files are updated. If any operation in the transaction fails, none of the files are updated.
base	The file revision on which two newer, conflicting file revisions are based.
binary file type	Perforce file type assigned to a nontext file. By default, the contents of each revision are stored in full, and the file is stored in compressed format.
branch	(*noun*) A codeline created by copying another codeline, as opposed to a codeline that was created by adding original files. *branch* is often used as a synonym for *branch view*. (*verb*) To create a codeline branch with `p4 integrate`.
branch form	The Perforce form you use to modify a branch.
branch mapping	Specifies how a branch is to be created by defining the location of the original codeline and the branch. The branch mapping is used by the integration process to create and update branches. Client workspaces, labels, and branch specifications cannot share the same name.
branch view	A specification of the branching relationship between two codelines in the depot. Each branch view has a unique name and defines how files are mapped from the originating codeline to the target codeline. See *branch*.
changelist	An atomic change transaction in Perforce. The changes specified in the changelist are not stored in the depot until the changelist is submitted to the depot.
changelist form	The Perforce form you use to modify a changelist.
changelist number	The unique numeric identifier of a changelist.
change review	The process of sending email to users who have registered their interest in changes made to specified files in the depot.
checkpoint	A copy of the underlying metadata at a particular moment in time. See *metadata*.

Term	Definition
client form	The Perforce form you use to define a client workspace.
client name	A name that uniquely identifies the current client workspace.
client root	The root directory of a client workspace. If two or more client workspaces are located on one machine, they cannot share a root directory.
client side	The right-hand side of a mapping within a client view, specifying where the corresponding depot files are located in the client workspace.
client workspace view	A set of mappings that specifies the correspondence between file locations in the depot and the client workspace.
client workspace	Directories on your workstation where you work on file revisions that are managed by Perforce. By default this name is set to the name of the machine on which your client workspace is located; to override the default name, set the `P4CLIENT` environment variable. Client workspaces, labels, and branch specifications cannot share the same name.
codeline	A set of files that evolve collectively. One codeline can be branched from another, allowing each set of files to evolve separately.
conflict	One type of conflict occurs when two users open a file for edit. One user submits the file, after which the other user can't submit because of a conflict. The cause of this type of conflict is two users opening the same file. The other type of conflict is when users try to merge one file into another. This type of conflict occurs when the comparison of two files to a common base yields different results, indicating that the files have been changed in different ways. In this case, the merge can't be done automatically and must be done by hand. The type of conflict is caused by nonmatching *diff*s. See *file conflict*.
counter	A numeric variable used by Perforce to track changelist numbers in conjunction with the review feature.
default changelist	The changelist used by Perforce commands, unless a numbered changelist is specified. A default pending changelist is created automatically when a file is opened for edit.
default depot	The depot name that is assumed when no name is specified. The default depot name is `depot`.
deleted file	In Perforce, a file with its head revision marked as deleted. Older revisions of the file are still available.
delta	The differences between two files.

Term	Definition
depot	A file repository hosted on the Perforce service. It contains all versions of all files ever submitted to the depot. There can be multiple depots on a single installation.
depot root	The root directory for a depot.
depot side	The left side of any client view mapping, specifying the location of files in a depot.
depot syntax	Perforce syntax for specifying the location of files in the depot.
detached	A workstation that cannot connect to the Perforce service.
diff	(*noun*) A set of lines that don't match when two files are compared. A *conflict* is a pair of unequal diffs between each of two files and a common third file. (*verb*) To compare the contents of files or file revisions.
donor file	The file from which changes are taken when propagating changes from one file to another.
exclusionary mapping	A view mapping that excludes specific files.
exclusionary access	A permission that denies access to the specified files.
file conflict	In a three-way file merge, a situation in which two revisions of a file differ from each other and from their base file. Also: an attempt to submit a file that is not an edit of the head revision of the file in the depot; typically occurs when another user opens the file for edit after you have opened the file for edit.
file pattern	Perforce command line syntax that enables you to specify files using wildcards.
file repository	The master copy of all files; shared by all users. In Perforce, this is called the *depot*.
file revision	A specific version of a file within the depot. Each revision is assigned a number, in sequence. Any revision can be accessed in the depot by its revision number, for example: `testfile#3`.
file tree	All the subdirectories and files under a given root directory.
file type	An attribute that determines how Perforce stores and diffs a particular file. Examples of file types are `text` and `binary`.
fix	A job that has been linked to a changelist.

Term	Definition
form	Screens displayed by certain Perforce commands. For example, you use the Perforce change form to enter comments about a particular changelist and to verify the affected files.
full-file storage	The method by which Perforce stores revisions of binary files in the depot: every file revision is stored in full. Contrast this with *reverse delta storage*, which Perforce uses for `text` files.
get	An obsolete Perforce term: replaced by *sync*.
group	A list of Perforce users.
have list	The list of file revisions currently in the client workspace.
head revision	The most recent revision of a file within the depot. Because file revisions are numbered sequentially, this revision is the highest-numbered revision of that file.
integrate	To compare two sets of files (for example, two codeline branches) and: • Determine which changes in one set apply to the other. • Determine if the changes have already been propagated. • Propagate any outstanding changes.
Inter-File Branching	Perforce's branching mechanism.
job	A user-defined unit of work tracked by Perforce. The job template determines what information is tracked. The template can be modified by the Perforce system administrator
job specification	A specification containing the fields and valid values stored for a Perforce job.
job view	A syntax used for searching Perforce jobs.
journal	A file containing a record of every change made to the Perforce service's metadata since the time of the last checkpoint.
journaling	The process of recording changes made to the Perforce service's metadata.
label	A named list of user-specified file revisions.
label view	The view that specifies which filenames in the depot can be stored in a particular label.
lazy copy	A method used by Perforce to make internal copies of files without duplicating file content in the depot. Lazy copies minimize the consumption of disk space by storing references to the original file instead of copies of the file.

Term	Definition
license file	Ensures that the number of Perforce users on your site does not exceed the number for which you have paid.
list access	A protection level that enables you to run reporting commands but prevents access to the contents of files.
local depot	Any depot located on the currently-specified Perforce service.
local syntax	The operating-system-specific syntax for specifying a filename.
lock	A Perforce file lock prevents other clients from submitting the locked file. Files are unlocked with the `p4 unlock` command or submitting the changelist that contains the locked file.
log	Error output from the Perforce service. By default, error output is written to standard error. To specify a log file, set the `P4LOG` environment variable or use the `p4d -L` flag when starting the service.
mapping	A single line in a view, consisting of a left side and a right side that specify the correspondences between files in the depot and files in a client, label, or branch. The left side specifies the depot files, and the right side specifies the client files. (See also *client workspace view*, *branch view*, *label view*).
MD5 checksum	The method used by Perforce to verify the integrity of archived files.
merge	The process of combining the contents of two conflicting file revisions into a single file.
merge file	A file generated by Perforce from two conflicting file revisions.
metadata	The data stored by the Perforce service that describes the files in the depot, the current state of client workspaces, protections, users, labels, and branches. Metadata includes all the data stored in the service except for the actual contents of the files.
modification time	The time a file was last changed.
nonexistent revision	A completely empty revision of any file. Syncing to a nonexistent revision of a file removes it from your workspace. An empty file revision created by deleting a file and the `#none` revision specifier are examples of nonexistent file revisions.
numbered changelist	A pending changelist to which Perforce has assigned a number.
open file	A file that you are changing in your client workspace.
owner	The Perforce user who created a particular client, branch, or label.

Term	Definition
`p4`	The Perforce Command-Line Client program, and the command you issue to execute Perforce commands from the operating system command line.
`p4d`	The program that runs the Perforce Service; `p4d` manages depot files and metadata.
P4Diff	A Perforce application that displays the differences between two files. P4Diff is the default application used to compare files during the file resolution process.
pending changelist	A changelist that has not been submitted.
Perforce service	The Perforce depot and metadata; also, the program that manages the depot and metadata.
protections	The permissions stored in the Perforce service's protections table.
RCS format	Revision Control System format. Used for storing revisions of text files. RCS format uses reverse delta encoding for file storage. Perforce uses RCS format to store text files. See also *reverse delta storage*.
read access	A protection level that enables you to read the contents of files managed by Perforce.
remote depot	A depot located on on a host other than that hosting the currently-specified Perforce service.
reresolve	The process of resolving a file after the file is resolved and before it is submitted
resolve	The process you use to reconcile the differences between two revisions of a file.
resource fork	One fork of a Mac file. (These files are composed of a resource fork and a data fork.) You can store resource forks in Perforce depots as part of an AppleSingle file by using Perforce's `apple` file type.
reverse delta storage	The method that Perforce uses to store revisions of text files. Perforce stores the changes between each revision and its previous revision, plus the full text of the head revision.
revert	To discard the changes you have made to a file in the client workspace.
review access	A special protections level that includes `read` and `list` accesses and grants permission to run the `p4 review` command.
review daemon	Any daemon process that uses the `p4 review` command. See also *change review*.
revision number	A number indicating which revision of the file is being referred to.

Term	Definition
revision range	A range of revision numbers for a specified file, specified as the low and high end of the range. For example, `myfile#5,7` specifies revisions 5 through 7 of `myfile`.
revision specification	A suffix to a filename that specifies a particular revision of that file. Revision specifiers can be revision numbers, change numbers, label names, date/time specifications, or client names.
service	In Perforce, the shared versioning service that responds to requests from Perforce applications. The Perforce service (`p4d`) maintains depot files and metadata describing the files and also tracks the state of client workspaces.
server root	The directory in which `p4d` stores its metadata and all the shared files. To specify the server root, set the `P4ROOT` environment variable.
shelving	The process of temporarily storing files in the Perforce service without checking in a changelist.
status	For a changelist, a value that indicates whether the changelist is new, pending, or submitted. For a job, a value that indicates whether the job is open, closed, or suspended. You can customize job statuses.
submit	To send a pending changelist and changed files to the Perforce service for processing.
subscribe	To register to receive email whenever changelists that affect particular files are submitted.
super access	An access level that gives the user permission to run *every* Perforce command, including commands that set protections, install triggers, or shut down the service for maintenance.
symlink file type	A Perforce file type assigned to symbolic links. On platforms that do not support symbolic links, symlink files appear as small text files.
sync	To copy a file revision (or set of file revisions) from the depot to a client workspace.
target file	The file that receives the changes from the donor file when you are integrating changes between a branched codeline and the original codeline.
text file type	Perforce file type assigned to a file that contains only ASCII text. See also *binary file type*.
theirs	The revision in the depot with which the client file is merged when you resolve a file conflict. When you are working with branched files, *theirs* is the donor file.
three-way merge	The process of combining three file revisions. During a three-way merge, you can identify where conflicting changes have occurred and specify how you want to resolve the conflicts.

Term	Definition
tip revision	In Perforce, the *head revision*. *Tip revision* is a term used by some other versioning systems.
trigger	A script automatically invoked by the Perforce service when changelists are submitted.
two-way merge	The process of combining two file revisions. In a two-way merge, you can see differences between the files but cannot see conflicts.
typemap	A Perforce table in which you assign Perforce file types to files.
user	The identifier that Perforce uses to determine who is performing an operation.
view	A description of the relationship between two sets of files. See *client workspace view*, *label view*, *branch view*.
wildcard	A special character used to match other characters in strings. Perforce wildcards are: • `*` matches anything except a slash • `...` matches anything including slashes • `%%0` through `%%9` used for parameter substitution in views
workspace	See *client workspace*.
write access	A protection level that enables you to run commands that alter the contents of files in the depot. `Write` access includes `read` and `list` accesses.
yours	The edited version of a file in the client workspace when you resolve a file. Also, the target file when you integrate a branched file.

Appendix

Perforce File Types

Perforce supports a set of file types that enable it to determine how files are stored by the Perforce service and whether the file can be diffed. When you add a file, Perforce attempts to determine the type of the file automatically: Perforce first determines whether the file is a regular file or a symbolic link, and then examines the first part of the file to determine whether it's **`text`** or **`binary`**. If any nontext characters are found, the file is assumed to be **`binary`**; otherwise, the file is assumed to be **`text`**. (Files in unicode environments are detected differently; see "Perforce file type detection and Unicode" on page 115.)

To determine the type of a file under Perforce control, issue the **`p4 opened`** or **`p4 files`** command. To change the Perforce file type, specify the **`-t`*`filetype`*** option. For details about changing file type, refer to the descriptions of **`p4 add`**, **`p4 edit`**, and **`p4 reopen`** in the *P4 Command Reference*.

Perforce file types

Perforce supports the following file types.

Keyword	Description	Comments	Stored as
`apple`	Mac file	AppleSingle storage of Mac data fork, resource fork, file type and file creator. For full details, please see the Mac client release notes.	full file, compressed, AppleSingle format
`binary`	Nontext file	Synced as binary files in the workspace. Stored compressed within the depot.	full file, compressed
`resource`	Mac resource fork	(Obsolete) This type is supported for backward compatibility, but the **`apple`** file type is recommended.	full file, compressed
`symlink`	Symbolic link	Perforce applications on UNIX, OS X, recent versions of Windows treat these files as symbolic links. On other platforms, these files appear as (small) text files.	delta
`text`	Text file	Synced as text in the workspace. Line-ending translations are performed automatically.	delta
`unicode`	Unicode file	Perforce services operating in unicode mode support the **`unicode`** file type. These files are translated into the local character set specified by **`P4CHARSET`**. Perforce services not in unicode mode do not support the **`unicode`** file type. For details, see the *Internationalization Notes*.	delta, UTF-8

Keyword	Description	Comments	Stored as
`utf16`	Unicode file	Whether the service is in unicode mode or not, files are transferred as UTF-8, and translated to UTF-16 (with byte order mark, in the byte order appropriate for the user's machine) in the client workspace. For details, see the *Internationalization Notes*.	delta, UTF-8

File type modifiers

You can apply file type modifiers to the base types of specific files to preserve timestamps, expand RCS keywords, specify how files are stored in the service, and more. For details about applying modifiers to file types, see "Specifying how files are stored in Perforce" on page 114.

The following table lists the file type modifiers.

Modifier	Description	Comments
`+C`	Perforce stores the full compressed version of each file revision	Default storage mechanism for **`binary`** files and newly-added **`text`**, **`unicode`**, or **`utf16`** files larger than 10MB.
`+D`	Perforce stores deltas in RCS format	Default storage mechanism for **`text`** files.
`+F`	Perforce stores full file per revision	For large ASCII files that aren't treated as text, such as PostScript files, where storing the deltas is not useful or efficient.
`+k`	RCS (Revision Control System) keyword expansion	Supported keywords are as follows: • `$Id$` • `$Header$` • `$Date$` Date of submission • `$DateUTC$` Date of submission in UTC time zone • `$DateTime$` Date and time of submission • `$DateTimeUTC$` Date and time of submission in UTC time zone. • `$DateTimeTZ$` Date and time of submission in the server's time zone, but including the actual time zone in the result.

Modifier	Description	Comments
		• `$Change$` • `$File$` • `$Revision$` • `$Author$` RCS keywords are case-sensitive. A colon after the keyword (for example, **`$Id:$`**) is optional.
`+ko`	Limited keyword expansion	Expands only the **`$Id$`** and **`$Header$`** keywords. Primarily for backwards compatibility with versions of Perforce prior to 2000.1, and corresponds to the **`+k`** (**`ktext`**) modifier in earlier versions of Perforce.
`+l`	Exclusive open (locking)	If set, only one user at a time can open a file for editing. Useful for binary file types (such as graphics) where merging of changes from multiple authors is not possible.
`+m`	Preserve original modification time	The file's timestamp on the local file system is preserved upon submission and restored upon sync. Useful for third-party DLLs in Windows environments, because the operating system relies on the file's timestamp. By default, the modification time is set to the time you synced the file.
`+S`	Only the head revision is stored	Older revisions are purged from the depot upon submission of new revisions. Useful for executable or **`.obj`** files.
`+S`*n*	Only the most recent ***n*** revisions are stored, where ***n*** is a number from 1 to 10, or 16, 32, 64, 128, 256, or 512.	Older revisions are purged from the depot upon submission of more than ***n*** new revisions, or if you change an existing **`+S`*n*** file's ***n*** to a number less than its current value. For details, see the *P4 Command Reference*.
`+w`	File is always writable on client	Not recommended, because Perforce manages the read-write settings on files under its control.

Modifier	Description	Comments
`+x`	Execute bit set on client	Used for executable files.
`+X`	Archive trigger required	The Perforce service runs an **`archive`** trigger to access the file. See the *Perforce Server Administrator's Guide: Fundamentals* for details.

Specifying how files are stored in Perforce

File revisions of binary files are normally stored in full within the depot, but only changes made to text files since the previous revision are normally stored. This approach is called *delta storage*, and Perforce uses RCS format to store its deltas. The file's type determines whether *full file* or *delta* storage is used.

Some file types are compressed to **`gzip`** format when stored in the depot. The compression occurs when you submit the file, and decompression happens when you sync (copy the file from the depot to your workspace). The client workspace always contains the file as it was submitted.

Warning! To avoid inadvertent file truncation, do not store binary files as **`text`**. If you store a binary file as text from a Windows machine and the file contains the Windows end-of-file character `^Z`, only the part of the file up to the `^Z` is stored in the depot.

Assigning File Types for Unicode Files

The Perforce service can be run in Unicode mode to activate support for filenames and Perforce metadata that contain Unicode characters, or in non-Unicode mode, where filenames and metadata must be ASCII, but textual files containing unicode content are still supported.

If you need to manage textual files that contain Unicode characters, but do not need Unicode characters in Perforce metadata, you do not need to run Perforce in Unicode mode. Assign the Perforce `utf16` file type to textual files that contain Unicode characters.

Your system administrator will be able to tell you which mode the service is using.

In either mode, Perforce supports a set of file types that enable it to determine how a file is stored and whether the file can be diffed. The following sections describe the considerations for managing textual files in Unicode environments.

To assign file type when adding a file to the depot, specify the `-t` option. For example:

```
p4 add -t utf16 newfile.txt
```

To change the file type of files in the depot, open the file for edit, specifying the `-t` option. For example:

```
p4 edit -t utf16 myfile.txt
```

Choosing the file type

When assigning file types to textual files that contain Unicode, consider the following:

- Do you need to edit and diff the files?

  Many IDEs create configuration files that you never edit manually or diff. To ensure they are never translated, assign such files the `binary` file type.

- Is your site managing files that use different character sets?

  If so, consider storing them using a `utf16` file type, to ensure they are not translated but still can be diffed.

Unicode mode services translate the contents of unicode files into the character set specified by `P4CHARSET`. The following table provides more details about how Unicode-mode services manage the various types of text files.

Text file type	Stored by Perforce as (unicode mode)	Validated?	Translated per `P4CHARSET`?	Translated per client platform
`text`	Extended ASCII	No	No	No
`unicode`	UTF-8	Yes (as UTF-16 and `P4CHARSET`)	Yes	No
`utf16`	UTF-8	Yes (as UTF-16)	No	No

Non-unicode-mode services do not translate or verify the contents of `unicode` files. Instead, the UTF-8 data is converted to UTF-16 using the byte order appropriate to the client platform. To ensure that such files are not corrupted when you edit them, save them as UTF-8 or UTF-16 from within your editing software.

Text file type	Stored by Perforce as (unicode mode)	Validated?	Translated per `P4CHARSET`?	Translated per client platform
`text`	Extended ASCII	No	No	No
`unicode`	UTF-8	Yes (as UTF-16 and `P4CHARSET`)	No	No
`utf16`	UTF-8	Yes (as UTF-16)	No	Yes

Perforce file type detection and Unicode

In both Unicode mode and non-Unicode mode, if you do not assign a file type when you add a file to the depot, Perforce (by default) attempts to detect file type by scanning the first 65536 characters of the file. If nonprintable characters are detected, the file is assigned the **binary** file type. (In Unicode mode, a further check is performed: if there are no nonprintable characters, and there are high-ASCII characters that are translatable using the character set specified by **P4CHARSET**, the file is assigned the **unicode** file type.)

Finally (for services running in Unicode mode or non-Unicode mode), if a UTF-16 BOM is present, the file is assigned the **utf16** file type. Otherwise, the file is assigned the **text** file type. (In Unicode mode, a further check is performed: files with high-ASCII characters that are undefined in the character set specified by **P4CHARSET** are assigned the **binary** file type.)

In most cases, there is no need to override Perforce's default file type detection. If you must override Perforce's default file type detection, you can assign Perforce file types according to a file's extension, by issuing the **p4 typemap** command. For more about using the typemap feature, refer to the *Perforce Server Administrator's Guide: Fundamentals*, and the *P4 Command Reference*.

Overriding file types

Some file formats (for example, Adobe PDF files, and Rich Text Format files) are actually **binary** files, but they can be mistakenly detected by Perforce as being **text**. To prevent this problem, your system administrator can use the **p4 typemap** command to specify how such file types are stored. You can always override the file type specified in the typemap table by specifying the **-t *filetype*** option.

Preserving timestamps

Normally, Perforce updates the timestamp when a file is synced. The modification time (**+m**) modifier is intended for developers who need to preserve a file's original timestamp. This modifier enables you to ensure that the timestamp of a file synced to your client workspace is the time on your machine when the file was submitted.

Windows uses timestamps on third-party DLLs for versioning information (both within the development environment and also by the operating system), and the **+m** modifier enables you to preserve the original timestamps to prevent spurious version mismatches. The **+m** modifier overrides the client workspace **[no]modtime** setting (for the files to which it is applied). For details about this setting, refer to "File type modifiers" on page 112.

Expanding RCS keywords

RCS (Revision Control System), an early version control system, defined keywords that you can embed in your source files. These keywords are updated whenever a file is committed to the repository. Perforce supports some RCS keywords.

To activate RCS keyword expansion for a file, use the **+k** modifier. RCS keywords are expanded as follows.

Keyword	Expands To	Example
`$Author$`	Perforce user submitting the file	`$Author: bruno $`

Keyword	Expands To	Example
`$Change$`	Perforce changelist number under which file was submitted	`$Change: 439 $`
`$Date$`	Date of last submission in format *YYYY/MM/DD*	`$Date: 2011/08/18 $`
`$DateTime$`	Date and time of last submission in format *YYYY/MM/DDhh:mm:ss* Date and time are as of the local time on the Perforce service at time of submission.	`$DateTime: 2011/08/18 23:17:02 $`
`$File$`	Filename only, in depot syntax (without revision number)	`$File: //depot/path/file.txt $`
`$Header$`	Synonymous with `$Id$`	`$Header: //depot/path/file.txt#3 $`
`$Id$`	Filename and revision number in depot syntax	`$Id: //depot/path/file.txt#3 $`
`$Revision$`	Perforce revision number	`$Revision: #3 $`

To display a file without expanding its keywords, use `p4 print -k` *filename*.

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/`).