# *Perforce 2007.2*
# *P4 User's Guide*

**May 2007**

# Table of Contents

# List of Examples

Preface

# About This Manual

This guide tells you how to use the Perforce Command-Line Client (`p4`). If you're new to SCM (software configuration management), 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 SCM.

## Command line versus GUIs

Perforce provides many client 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 of your Perforce server.

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

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, 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, for details.

5. Learn to refine your client view. See "Refining client views" on page 24 for details.

These basic skills enable you to do much of your daily work. Other tasks involving code base maintenance (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 *Perforce 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 client programs, see our documentation web page, available from our web site at `http://www.perforce.com`.

| For specific information about... | See this documentation |
|---|---|
| The basics of Perforce | *Introducing Perforce* |
| Installing and administering the Perforce server, the proxy server, and security settings | *Perforce System Administrator's Guide* |
| `p4` command line flags and options (reference) | *Perforce Command Reference*, `p4 help` |
| P4V, the cross-platform Perforce Visual Client | *Getting Started with P4V*, P4V online help |
| P4Web, the browser-based Perforce client application | *How to use P4Web*, P4Web online help |
| P4Win, the Perforce Windows GUI | *Getting Started with P4Win*, P4Win online help |
| Perforce plug-ins | IDEs: *Using IDE Plug-ins*<br>Others: online help from the Perforce menu |
| Developing Perforce client applications using the Perforce C/C++ API | *C/C++ API User's Guide* |

## 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`.

# Chapter 1 Installing P4

This chapter tells you how to install the Perforce Command-Line Client (`p4`) on a client machine. For details about installing the Perforce Server, refer to the *Perforce System Administrator's Guide*.

## Installing P4 on UNIX and OS X

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

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

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

   The Perforce client programs are typically downloaded to `/usr/local/bin`.

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

3. Configure the server 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.)

## 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/perforce/loadprog.html`

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 you have a server 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: 10.0.0.196:2345
Server address: ida:1818
Server root: /usr/depot/p4d
Server date: 2005/06/13 08:46:34 -0700 PDT
Server version: P4D/FREEBSD4/2005.1/79540 (2005/05/10)
Server license: Perforce Software 200 users (expires 2007/01/31)
```

If your configuration settings are incorrect, the following message is displayed::

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

# Chapter 2 Configuring P4

This chapter tells you how to configure connection settings.

## Configuration overview

Perforce uses a client/server architecture: you *sync* files from the server repository, called the *depot*, and edit them on your client machine in your *client workspace*. This chapter assumes that your system administrator has a Perforce server running. For details about installing the Perforce Server, refer to the *Perforce System Administrator's Guide*.

To set up your client workspace so you can work with the server, perform the following steps.

| Step# | For details, see... |
|---|---|
| 1. Configure settings for your server host and port (to specify where the Perforce Server is running). | "Configuring Perforce settings" on page 19. |
| 2. Define your client workspace (at a minimum, assign a name and specify a workspace root where you want local copies of depot files stored). | "Defining client workspaces" on page 22. |
| 3. Verify the connection. | "Verifying connections" on page 23. |

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 48 and the description of the `p4 sync` command in the *Perforce Command Reference*.

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

### What is a client workspace?

A Perforce *client workspace* is a set of directories on the client machine 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 Server. If no workspace name is specified (by setting the `P4CLIENT` environment variable) the default workspace name is the name of the client machine. To specify the effective workspace name, set the `P4CLIENT` environment variable. A client machine can contain multiple workspaces.

All files within a Perforce client workspace share a root directory, called the *client root*. The client 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 client views do not map depot files to overlapping locations on the client machine.

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.

## 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 client workspace is tracked and managed by the Perforce server. To avoid conflicts with the file management that is performed by the server, do not manually change read-only permission settings on files. You can verify that the state of the client workspace corresponds to the server's record of that state. For details, refer to Technote 2 on the Perforce web site:

```
http://www.perforce.com/perforce/technotes/note002.html
```

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 client views" on page 24.

# Configuring Perforce settings

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

1. On the command line, using flags
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, in the Perforce user registry (set by issuing the `p4 set` command)
6. On Windows, in the Perforce system registry (set by issuing the `p4 set -s` command)

To configure your client machine to connect to a Perforce server, you specify the name of the host where the server is running, and the port on which the server is listening. The default server host is `perforce` and default server port is 1666. If the server is running on your client machine, specify `localhost` as the host name. If the server 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 detached (without a connection to a Perforce server), see "Working detached" on page 57.

## Using the command line

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

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

Server settings specified on the command line override any settings specified in config files, environment variables, or the Windows registry. For more details about command-line flags, refer to the discussion of global options in the *Perforce 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 flags).

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. |
| `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 client command. |
| `P4PORT` | The host and port number of the Perforce server or proxy with which to communicate. |
| `P4USER` | Current Perforce user name. |

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

**Example:** *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 server at* `ida:1818`. *The second workspace is called* `ona-agave`. *Its client root is* `/home/ona/p4-ona`, *and it uses the Perforce server at* `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=ida:1818
P4CLIENT=ona-ash
```

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

```
P4PORT=warhol:1666
P4CLIENT=graphicwork
```

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

## Using environment variables

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

| If the server is running on | and is listening to port | set P4PORT to |
|---|---|---|
| your computer | `1666` | `localhost:1666` |
| `perforce` | `1666` | `1666` |
| `houston` | `3435` | `houston:3435` |
| `deneb.com` | `1818` | `deneb.com:1818` |

## Using the Windows registry

On Windows machines, you can store connection settings in the registry by issuing the `p4 set` command. For example:

```
p4 set P4PORT=tea:1667
```

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

- `p4 set` *`setting=value`*: for the current Windows login.
- `p4 set -s` *`setting=value`*: for all users on the local machine. Overrides any registry settings made for the local user. Requires Perforce admin privilege.

To see which settings are in effect, type the `p4 set` command. For details about the `p4 set` command, see the *Perforce 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 specification form in your text editor. (For details about Perforce forms, refer to "Using Perforce forms" on page 45.)
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 48. For details about relocating files on your machine, see "Changing the location of your workspace" on page 30.

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

- **Workspace name**

  The workspace name defaults to the client machine's hostname, but a client machine can contain multiple workspaces. To specify the effective workspace, set `P4CLIENT`.
- **Client root**

  The client root is the top directory of your client workspace, where Perforce stores your working copies of depot files. Be sure to set the client root, or you might inadvertently sync files to your client machine's root directory.

Your *client view* determines which files in the depot are mapped to a client workspace and enables the server 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 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 client views" on page 24.

**Example:** *Setting the client view*

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

```
Client:  bruno_ws
Update:  2004/11/29 09:46:53
Access:  2005/03/02 10:28:40
Owner:   bruno
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
Client address: 127.0.0.1:28
Server address: localhost:1667
Server root: c:\p4root
Server date: 2005/03/02 11:16:11 -0800 Pacific Standard Time
Server version: P4D/NTX86/2005.1/75548 (2005/02/07)
Server license: 100 clients 100 users (expires 206/09/06)
```

The `Server address:` field shows the Perforce server to which `p4` connected and also displays the host and port number on which the Perforce server 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. If the value is anything else, `P4PORT` is set incorrectly.

# Refining client 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 client view, issue the `p4 client` command. Perforce displays the client 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 view. For more information, see the `p4 client` command description and the description of views in the *Perforce 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 view mapping is called the *depot side*, and the right-hand side is the *client side*.

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

To map groups of files in client 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 client view mappings.

| Wildcard | Description |
|---|---|
| `*` | Matches anything except slashes. Matches only within a single directory. Case sensitivity depends on your server 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 view:

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

all files in the depot's `dev` branch are mapped to the corresponding locations in the workspace. For example, the file `//depot/dev/main/jam/Makefile` is mapped to the client 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 client 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 client 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 client view also ensures that your commands do not inadvertently affect the

entire depot. To restrict the client view, change the left-hand side of the `View:` field to specify the relevant portion of the depot.

**Example:** *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:*

```
//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:** *Multiple mappings in a single client 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/...
```

## Mapping files to different filenames

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

**Example:** *Files with different names in the depot and client 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:** *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 client 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:** *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 client view looks like this:*

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

## 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:** *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 client workspace. To overlay the contents of a second directory in your client workspace, use a plus sign (+) in front of the mapping.

**Example:** *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.

## Mapping Windows workspaces across multiple drives

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

```
Client:     bruno_ws
Update:     2004/11/29 09:46:53
Access:     2005/03/02 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 client workspace on multiple machines with different platforms, delete the `Host:` entry and set the `AltRoots:` field in the client specification. You can specify a maximum of two alternate client 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 client 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 client 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 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, refer to the *Perforce System Administrator's Guide*.

## 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.
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 client 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, the Perforce server makes unopened files read-only. To avoid inadvertently overwriting changes or causing syncs to fail, specify `noallwrite`. | `noallwrite` |
| `[no]clobber` | Specifies whether `p4 sync` overwrites writable but unopened workspace files. (By default, Perforce does not overwrite unopened files if they are writable.) | `noclobber` |
| `[no]compress` | Specifies whether data is compressed when it is sent between the client and the server. | `nocompress` |

| Option | Description | Default |
|---|---|---|
| `[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) flag.<br>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:<br>• 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.<br>• If `nomodtime` is set, the modification date is the date and time of sync, regardless of Perforce client version.<br>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).<br>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.<br>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. |
| `revertunchanged` | Only those files with content or type changes are submitted to the depot. Unchanged files are reverted. |
| `revertunchanged+reopen` | Only those files with content or type 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 or type changes are submitted to the depot. Any unchanged files are moved to the default changelist. |
| `leaveunchanged+reopen` | Only those files with content or type changes are submitted to the depot. Unchanged files are moved to the default changelist, and changed files are reopened in the default changelist.<br>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` | Macintosh pre-OS X: `CR` only |
| `win` | Windows- style: `CR`, `LF` |
| `share` | Line endings are `LF` with any `CR/LF` pairs translated to `LF`-only style before storage or syncing with the depot.<br>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.<br>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, the `share` option eliminates problems caused by Windows-based editors that insert carriage returns in text files. |

For detailed information about how Perforce uses the line-ending settings, see Tech Note 63 on the Perforce web site:

```
http://www.perforce.com/perforce/technotes/note063.html
```

## Deleting client workspace specifications

To delete a client workspace specification, issue the `p4 client -d` *`clientname`* command. Deleting a client specification removes the Perforce server'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 changelists that have been opened from 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 server to require passwords and to impose a connection time limit. The following sections provide details.

## Passwords

Depending on the security level at which your Perforce server 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` flag 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.

Your system administrator can configure your Perforce server to require "strong" passwords. A password is considered strong if it is at least eight characters long and contains at least two of the following:

- Uppercase letters
- Lowercase letters
- Non-alphabetic characters

For example, `a1b2c3d4`, `A1B2C3D4`, `aBcDeFgH` are 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 System Administrator's Guide* 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 the server 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`* flag when you issue `p4` commands (for instance, `p4 -P` *`mypassword`* `submit`)
- Windows: store your password in the registry 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 server 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 flags, 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 for your server, 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 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 *Perforce Command Reference*.

# Chapter 3 Issuing P4 Commands

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

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

## Command-line syntax

The basic syntax for commands is as follows:

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

The following flags 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` |
| `-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` |
| `-p server` | Specifies the Perforce server's host and port number, overriding `P4PORT`. |
| | `p4 -p deneb:1818 clients` |
| `-P password` | Supplies a Perforce password, overriding `P4PASSWD`. Usually used in combination with the `-u username` flag. |
| | `p4 -u earl -P secretpassword job` |
| `-s` | Prepend a tag to each line of output (for scripting purposes). |
| | `p4 -s info` |

| Global options | Description and Example |
|---|---|
| `-u username` | Specifies a Perforce user, overriding `P4USER`.<br>`p4 -u bill user` |
| `-x filename` | Read arguments, one per line, from the specified file. To read arguments from standard input, specify "`-x -`".<br>`p4 -x myargs.txt` |
| `-V` | Displays the version of the `p4` executable. |

To display the flags 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# ] [ -t filetype ] file ...
        Open a new 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. The file must
        either not exist in the depot, or it must be deleted at the
        current head revision. Files may be deleted and re-added.
[...]
```

For the full list of global options, commands, and command-specific flags, see the *Perforce 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:** *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 server 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 |
| `//...` | All files in all depots |

The `*` wildcard is expanded locally by the operating system before the command is sent to the server. 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 server, and, because the server 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 the Perforce Server.

## 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 server.

### Length limitations

Names assigned to Perforce objects such as branches, client workspaces, and so on, cannot exceed 1024 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) |
| `%` | Wildcard |
| `/` | 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
```

### 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 *Perforce 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`*<br>Example:<br>`p4 sync //depot/dev/main/jam/Jambase#3`<br>Refers to revision 3 of file `Jambase` |
| The revision submitted as of a specified changelist | *`file@changelist_number`*<br>Examples:<br>`p4 sync //depot/dev/main/jam/Jambase@126`<br>Refers to the version of `Jambase` when changelist 126 was submitted, even if it was not part of the change.<br>`p4 sync //depot/...@126`<br>Refers to the state of the entire depot at changelist 126 (numbered changelists are explained in “Managing changelists” on page 52). |
| The revision in a specified label | *`file@labelname`*<br>Example:<br>`p4 sync //depot/dev/main/jam/Jambase@beta`<br>The revision of `Jambase` in the label called `beta`. For details about labels, refer to “Using labels” on page 76. |
| The revision last synced to a specified client workspace | *`file@clientname`*<br>Example:<br>`p4 sync //depot/dev/main/jam/Jambase@bruno_ws`<br>The revision of `Jambase` last synced to client workspace `bruno_ws` |

| Revision needed | Syntax and example |
|---|---|
| Remove the file | *file*#none<br>Example:<br>p4 sync //depot/dev/main/jam/Jambase#none<br>Removes Jambase from the client workspace. |
| The most recent version of the file | *file*#head<br>Example:<br>p4 sync //depot/dev/main/jam/Jambase#head<br>Same as p4 sync //depot/dev/main/jam/Jambase<br>(If you omit the revision specifier, the head revision is synced.) |
| The revision last synced to your workspace | *file*#have<br>Example:<br>p4 files //depot/dev/main/jam/Jambase#have |
| The head revision of the file in the depot on the specified date | *file@date*<br>Example:<br>p4 sync //depot/dev/main/jam/Jambase@2005/05/18<br>The head revision of Jambase as of midnight May 18, 2005. |
| The head revision of the file in the depot on the specified date at the specified time | *file*@"*date*[:*time*]"<br>Example:<br>p4 sync //depot/dev/main/jam/Jambase@"2005/05/18"<br>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<br>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:** *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:** *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 Perforce server. To display the date, time, offset from GMT, and time zone in effect at your Perforce server, issue the `p4 info` command. The Perforce server stores times as the number of seconds since 00:00:00 GMT Jan. 1, 1970), so if you move your server across time zones, the times stored on the server 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 changes`
- `p4 files`
- `p4 integrate`
- `p4 jobs`
- `p4 print`
- `p4 sync`

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:** *Listing changes using revision ranges*

*A release manager needs to see a quick list of all changes made to the jam project in July 2000. He types:*

```
p4 changes //depot/dev/main/jam/...@2000/7/1,2000/8/1
```

*The resulting list of changes looks like this:*

```
Change 673 on 2000/07/31 by bruno@bruno_ws 'Final build for QA'
Change 633 on 2000/07/1 by bruno@bruno_ws 'First build w/bug fix'
Change 632 on 2000/07/1 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 flags 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 client machine | `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 *Perforce 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.<br>2. Open the file for add in a changelist (`p4 add`).<br>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`).<br>2. Open the file for edit in a changelist (`p4 edit`).<br>3. Make your changes.<br>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.<br>2. Submit the changelist (`p4 submit`). The file is deleted from the depot. |

| Task | Description |
|---|---|
| Discarding changes | Revert the files or the changelist in which the files are open. Reverting has the following effects on open files:<br>Add: no effect—the file remains in your workspace.<br>Edit: the revision you opened is resynced from the depot, overwriting any changes you made to the file in your workspace.<br>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 38. 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 client views" on page 24.

**Example:** *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 42.

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 39.

The Perforce server tracks the revisions that you sync (in a database located on the server machine). For maximum efficiency, Perforce does not resync an already-synced file revision. To resync files you (perhaps inadvertently) deleted manually, specify the `-f` flag 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 Tech Note 12 on the Perforce web site:

```
http://www.perforce.com/perforce/technotes/note012.html
```

**Example:** *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:** *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>
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.

## 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:** *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, the Perforce server 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:** *Reverting a file*

*Bruno decides not to add his text files after all.*

```
C:\bruno_ws\dev\main\docs\manuals>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` flag 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:** *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` flag. 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` flag

The Perforce server 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`

To submit a numbered changelist, specify the `-c` flag when you issue the `p4 submit` command. To submit the default changelist, omit the `-c` flag. For details, refer to the `p4 submit` command description in the *Perforce 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`.

## 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:** *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, the Perforce server might renumber it, as shown in the following example.

**Example:** *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 server, and the server updates the files in the depot. 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` flag.

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 the Perforce server's management of your workspace, do not change file write permissions manually.

## 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 System Administrator's Guide* for more information.

## Renaming and moving files

To rename or move files, you use the `p4 delete` and `p4 integrate` commands to simultaneously create the new file and delete the original file, thereby preserving its revision history. The process is as follows:

```
p4 integrate source_file target_file
p4 delete source_file
p4 submit
```

To rename groups of files, use matching wildcards in the *`source_file`* and *`target_file`* specifiers. To rename files, you must have Perforce `write` permission for the specified files. (For details about Perforce permissions, see the *Perforce System Administrator's Guide.*)

When you rename or move a file using `p4 integrate`, the Perforce server 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 73.

## 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 and submitted 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` flag to exclude the file diffs.) |

For more information, see "Changelist reporting" on page 95.

## Diffing files

Perforce provides a program that enables you 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 the file.

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 diff on your client machine, but the `p4 diff2` command performs the diff on the server machine and sends the results to your client machine.

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`*<br>or<br>`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` |
| All files in release 1 | All files in release 2 | `p4 diff2 //depot/rel1/... //depot/rel2/...` |

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

# Working detached

If you need to work detached (without access to your Perforce server) on files under Perforce control, you must reconcile your work with the Perforce server when you regain access to the server. The following method for working detached assumes that you work on files in your client workspace or update the workspace with your additions, changes, and deletions before you update the depot.

For platform-specific details about working detached, see Tech Note 2 on the Perforce web site:

```
http://www.perforce.com/perforce/technotes/note002.html
```

To work detached:

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 diff` to find all files in your workspace that have changed. (You need to track new files manually.)
3. Update the depot by opening files for add, edit, or delete as required and submitting the resulting changelists.

The following sections provide more details.

## Finding changed files

To detect changed files, issue the `p4 diff` command. The following flags enable you to locate files that you changed or deleted manually, without opening them for edit or delete in Perforce.

| Flag | Description |
|---|---|
| `p4 diff -se` | Lists workspace files that are not open for edit but have been changed since being synced. To update the depot with these files, open them for edit and submit them. |
| `p4 diff -sd` | Lists files that are not open for delete but have been manually deleted from the workspace. To update the depot with these file deletions, open them for delete and submit them. |

## Submitting your changes

To update the depot with the changes that you made to the client workspace while working detached, use the `p4 diff` flags described above with the `-x` flag, as shown in the following examples. The `-x` flag directs the `p4 edit` command to accept arguments from the pipe (or a file).

To open changed files for edit after working detached, issue the following command:

```
p4 diff -se | p4 -x - edit
```

To delete files from the depot that were removed from the client workspace, issue the following command:

```
p4 diff -sd | p4 -x - delete
```

Open any new files for add; then submit the changelist containing your additions, changes, and deletions.

# 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 67.

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, the Perforce server 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 56.

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 61 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). |

| File revision | Description |
|---|---|
| *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 62)
- Edit the merge file with a text editor (see "Editing the merge file" on page 63)
- Merge changes selectively using a merge program (see "Merging to resolve conflicts" on page 63)

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 flags" on page 66.

To reresolve a resolved but unsubmitted file, specify the `-f` flag 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`*.<br>• If *`yours`* is identical to *`base`*, accept *`theirs`*.<br>• If *`yours`* and *`theirs`* are different from *`base`*, and there are no conflicts between *`yours`* and *`theirs`*; accept *`merge`*.<br>• 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`*. |

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 `ay`.

By default, only the conflicts between the *`yours`* and *`theirs`* files are marked. To generate difference markers for all differences, specify the `-v` flag 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:<br>• If *`theirs`* is identical to *`base`*, accept *`yours`*.<br>• If *`yours`* is identical to *`base`*, accept *`theirs`*.<br>• If *`yours`* and *`theirs`* are different from *`base`*, and there are no conflicts between *`yours`* and *`theirs`*; accept *`merge`*.<br>• 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. |

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

**Note** The *merge* file is generated by the Perforce server, but the differences displayed by dy, dt, dm, and d are generated by the client machine'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 56.

**Example:** *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 67.

## Resolve command-line flags

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

| Flag | 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:<br>• If *`theirs`* is identical to *`base`*, accept *`yours`*.<br>• If *`yours`* is identical to *`base`*, accept *`theirs`*.<br>• If *`yours`* and *`theirs`* are different from *`base`*, and there are no conflicts between *`yours`* and *`theirs`*, accept *`merge`*.<br>• 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:<br>• If *`theirs`* is identical to *`base`*, accept *`yours`*.<br>• If *`yours`* is identical to *`base`*, accept *`theirs`*.<br>• Otherwise skip this file. |

**Example:** *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.<br>When you diff depot files, the Perforce server 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 *Perforce 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 and Branching

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

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.

# Branching

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 server 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 73.

## 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 a Macintosh driver, using the UNIX code as their starting point.

  *Solution*: create a Macintosh 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, the Perforce server 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 specification* and use it as an argument when you issue the `p4 integrate` command. To create a branch specification, 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 specification has no effect on any files in the depot or client workspace. The branch specification merely maps source files to target files.

To use the branch specification 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 specification 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 specification, issue the following command:

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

To delete a branch specification, issue the `p4 branch -d` *`branchname`* command . Deleting a branch specification 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:** *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` command, as follows:

1. Issue the `p4 integrate` command to schedule the files for resolve.
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:** *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` command, you must have Perforce `write` permission on the target files, and `read` access on the source files. (See the *Perforce System Administrator's Guide* 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 `-d` or `-D` flags. For details, refer to the *Perforce Command Reference*.

## Integrating using branch specifications

To integrate changes from one set of files and directories to another, you can use a branch specification when you issue the `p4 integrate` command. The basic syntax of the integrate command when using a branch specification 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 specification, specify the `-r` flag. This flag enables you to integrate in either direction between two branches without requiring you to create a branch specification for each direction.

**Example:** *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. To integrate between unrelated files, specify the `-i` flag. 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. The *`base`* file is the common ancestor.

**Example:** *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 20 revisions since the fixes were submitted, Earl knows that the bug fixes he wants were made to file revisions submitted in changelist 30. 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 revision of Bruno's file that was submitted in changelist 30 is scheduled for resolve. That is, Earl sees only the*

*changes that Bruno made to* `scan.c` *in changelist 30. The file revision that was present in the depot at changelist 29 is used as the* `base` *file.*

### Reintegrating and reresolving files

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

A target that has been resolved but not submitted can be resolved again by specifying the `-f` flag 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 flag (`-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]` |

## Using 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 the Perforce server.

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 by the `tag` command 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 (`@2003/07/01`).

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:** *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:    2005/03/07 13:07:39
Access:    2005/03/07 13:13:35
Owner:    earl
Description:
     Created by earl.
Options:     unlocked
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:** *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:    2005/03/07 13:07:39
Access:    2005/03/07 13:13:35
Owner:    earl
Description:
     Created by earl.
Options:     unlocked
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 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
```

## 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:** *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 nightly20061201` *and uses the label's* `Revision:` *field to automatically tag all files as of changelist 1234 with the* `nightly20061201` *label:*

```
Label:    nightly20061201
Owner:    earl
Description:
     Nightly build process.
Options:     unlocked
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:** *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 patch20061201` *and uses the label's* `Revision:` *field to automatically tag only those files submitted in changelist 1238 with the* `patch20061201` *label:*

```
Label:    patch20061201
Owner:    earl
Description:
     Patch to 2006/12/01 nightly build.
Options:     unlocked
View:
     //depot/...
Revision:
     @1238,1238
```

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

**Example:** *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
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 System Administrator's Guide*.

# Chapter 7 Defect Tracking

A *job* is a numbered (or named) work request managed by the Perforce server. 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 System Administrator's Guide*.

If you want to integrate Perforce with your in-house defect tracking system, or develop an integration with a third-party defect tracking system, see the P4DTI product information page on the Perforce web site.

## 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:** *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:     2005/11/14 17:12:21
Description:
     MAXLINE on NT can't account for NT 4.0 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 (`job`*`nnnnnn`*). | Last job number + 1 |

| Field Name | Description | Default |
|---|---|---|
| `Status` | • `open:` job has not yet been fixed.<br>• `closed:` job has been completed.<br>• `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. |
| `Date` | The date the job was last modified. | Updated by the Perforce server 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:** *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:** *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:** *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:** *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.

| | |
|---|---|
| `=` | Equal |
| `>` | Greater than |
| `<` | Less than |
| `>=` | Greater than or equal to |
| `<=` | Less than or equal to |

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.<br>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.<br>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, `2005/07/15:13:21:40` | Dates are matched chronologically. If a time is not specified, the operators `=`, `<=`, and `>=` match the whole day. |

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:** *Using dates within expressions*

*Bruno wants to view all jobs modified on July 13, 2005. He enters*

```
p4 jobs -e 'ModifiedDate=2005/07/13'
```

## 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 System Administrator's Guide* 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:** *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:   2005/06/02 13:11:57
Access:   2005/06/03 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:** *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 8 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 *Perforce Command Reference*, or issue the `p4 help` command.

## Common flags used in scripting and reporting

The following command-line flags enable you to specify settings on the command line and in scripts. For full details, refer to the description of global options in the *Perforce Command Reference*.

| Flag | Description |
|---|---|
| `-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 server:port` | Specifies the host and port number of the Perforce server. |
| `-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:<br>`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` flag to write the form to standard output and the `-i` flag 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>/Crash when exiting./' 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>/Crash when exiting./' | 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 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` |
| Currently synced files | `p4 have` |
| The contents of specified files | `p4 print` |

| To display this information | Use this command |
|---|---|
| The mapping of files' depot locations to the corresponding workspace locations. | `p4 where` |
| 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 | `p4 files //depot/...` |
| 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. | |
| The files currently synced to the specified client workspace | `p4 files @clientname` |
| The files mapped by your client workspace view | `p4 files //clientname/...` |
| 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 2001/11/13 by earl@earl-dev-yew (text)
'Handle platform variants better'
... #34 change 598 edit on 2001/10/24 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 2001/10/03 by gale@gale-jam-oak (text)
'Version strings & release notes'
```

To display the entire description of each changelist, specify the `-l` flag.

## 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 -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` flag. 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` flag.

**Example:** *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 System Administrator's Guide* and to the description of the `p4 review` command in the *Perforce 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 changelist known to the system. The following table lists command-line flags 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`<br>or<br>`p4 changes -s submitted` |
| 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` flag. 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 flag 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 97.

## 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` |
| 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 84 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` |
| 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 2005/03/07
dai <dai@p4demo.com> (Dai Sato) accessed 2005/03/04
earl <earl@p4demo.com> (Earl Ashby) accessed 2005/03/07
gale <gale@p4demo.com> (Gale Beal) accessed 2001/06/03
hera <hera@p4demo.com> (Hera Otis) accessed 2001/10/03
ines <ines@p4demo.com> (Ines Rios) accessed 2005/02/02
jack <jack@submariner> (jack) accessed 2005/03/02
mei <mei@p4demo.com> (Mei Chang) accessed 2001/11/14
ona <ona@p4demo.com> (Ona Birch) accessed 2001/10/23
quinn <quinn@p4demo.com> (Quinn Cass) accessed 2005/01/27
raj <raj@p4demo.com> (Raj Bai) accessed 2001/07/28
vera <vera@p4demo.com> (Vera Cullen) accessed 2005/01/15
```

## 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 2005/03/07 root c:\bruno_ws ''
Client dai-beos-locust 2002/10/03 root /boot/home/src ''
Client earl-beos-aspen 2002/04/15 root /boot/src ''
Client earl-dev-beech 2002/10/26 root /home/earl ''
Client earl-dev-guava 2002/09/08 root /usr/earl/develoment ''
Client earl-dev-yew 2004/11/19 root /tmp ''
Client earl-mac-alder 2002/03/19 root Macintosh HD:earl ''
Client earl-os2-buckeye 2002/03/21 root c:\src ''
Client earl-qnx-elm 2001/01/17 root /src ''
Client earl-tupelo 2001/01/05 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`, or `spec`), 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 server, see the *Perforce System Administrator's Guide*.

## 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:** *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 A 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 server. |
| apple file type | Perforce file type assigned to Macintosh 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*.<br>(*verb*) To create a codeline branch with `p4 integrate`. |
| branch form | The Perforce form you use to modify a branch. |
| branch specification | Specifies how a branch is to be created by defining the location of the original codeline and the branch. The branch specification 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*. |

| Term | Definition |
|---|---|
| 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 server metadata at a particular moment in time. See *metadata*. |
| 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 the client machine where you work on file revisions that are managed by Perforce. By default this name is set to the name of the host machine on which the 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. |

| Term | Definition |
|---|---|
| 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.<br>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.<br>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. |
| depot | A file repository on the Perforce server. It contains all versions of all files ever submitted to the depot. There can be multiple depots on a single server. |
| 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 client machine that cannot connect to a Perforce server. |
| 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.<br>(*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. |

| Term | Definition |
|---|---|
| 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.<br><br>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. |
| 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. |

| Term | Definition |
|---|---|
| integrate | To compare two sets of files (for example, two codeline branches) and:<br>• Determine which changes in one set apply to the other.<br>• Determine if the changes have already been propagated.<br>• Propagate any outstanding changes. |
| Inter-File Branching | Perforce's proprietary 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 server's metadata since the time of the last checkpoint. |
| journaling | The process of recording changes made to the Perforce server'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. |
| 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 current Perforce server. |
| 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. |

| Term | Definition |
|---|---|
| log | Error output from the Perforce server. 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 server. |
| 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.<br>(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 server that describes the files in the depot, the current state of client workspaces, protections, users, clients, labels, and branches. Metadata includes all the data stored in the server 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. |
| `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 on the Perforce server that manages the depot and the 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. |

| Term | Definition |
|---|---|
| P4Win | The Perforce Windows Client, a Windows application that enables you to perform Perforce operations and view results graphically. |
| pending changelist | A changelist that has not been submitted. |
| Perforce server | The Perforce depot and metadata on a central host. Also the program that manages the depot and metadata. |
| protections | The permissions stored in the Perforce server'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 a server other than the current Perforce server. |
| 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 Macintosh file. (Macintosh 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. |
| server | In Perforce, the program that executes the commands sent by client programs. The Perforce server (`p4d`) maintains depot files and metadata describing the files and also tracks the state of client workspaces. |
| server root | The directory in which the server program stores its metadata and all the shared files. To specify the server root, set the `P4ROOT` environment variable. |
| 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 server 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 server for maintenance. |
| symlink file type | A Perforce file type assigned to UNIX symbolic links. On non-UNIX clients, symlink files are stored as 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. |

| Term | Definition |
|---|---|
| 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. |
| tip revision | In Perforce, the *head revision*. *Tip revision* is a term used by some other SCM systems. |
| trigger | A script automatically invoked by the Perforce server 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:<br>• `*` matches anything except a slash<br>• `...` matches anything including slashes<br>• `%%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 B Perforce File Types

Perforce supports a set of file types that enable it to determine how files are stored by the Perforce server 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 of type `unicode` are detected only when the server is running in Unicode mode; for details, see your system administrator.)

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`* flag. For details about changing file type, refer to the descriptions of `p4 add`, `p4 edit`, and `p4 reopen` in the *Perforce Command Reference*.

## Perforce file types

Perforce supports the following file types.

| Keyword | Description | Comments | How stored by the Perforce server |
|---|---|---|---|
| `apple` | Macintosh file | AppleSingle storage of Macintosh data fork, resource fork, file type and file creator.<br>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` | Macintosh resource fork | (Obsolete) This type is supported for backward compatibility, but the `apple` file type is recommended. | full file, compressed |
| `symlink` | Symbolic link | UNIX and BeOS client machines treat these files as symbolic links. Non-UNIX client machines treat them as text files. | delta |
| `text` | Text file | Synced as text in the workspace. Line-ending translations are performed automatically. | delta |

| Keyword | Description | Comments | How stored by the Perforce server |
|---|---|---|---|
| `unicode` | Unicode file | Perforce servers operating in unicode mode support the `unicode` file type. These files are translated into the local character set.<br>For details, see the *Internationalization Notes*. | UTF-8, UTF-16, or UTF-32 |
| `utf16` | Unicode file | If the Perforce server is operating in internationalized mode, files are translated into the local character set.<br>If the Perforce server is operating in non-internationalized mode, files are transferred as UTF-8, and translated to UTF-16 (with byte order mark, in client byte order) in the client workspace.<br>For details, see the *Internationalization Notes*. | 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 on the server, and more. For details about applying modifiers to file types, see"Specifying how files are stored in the server" on page 114.

The following table lists the file type modifiers.

| Modifier | Description | Comments |
|---|---|---|
| `+C` | Server stores the full compressed version of each file revision | Default server storage mechanism for `binary` files. |
| `+D` | Server stores deltas in RCS format | Default server storage mechanism for `text` files. |
| `+F` | Server 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. |

| Modifier | Description | Comments |
|---|---|---|
| `+k` | RCS (Revision Control System) keyword expansion | Supported keywords are:<br>• `$Author$`<br>• `$Change$`<br>• `$Date$`<br>• `$DateTime$`<br>• `$File$`<br>• `$Header$`<br>• `$Id$`<br>• `$Revision$`<br>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.<br>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 on the server | Older revisions are purged from the depot upon submission of new revisions. Useful for executable or `.obj` files. |
| `+Sn` | Only the most recent *n* revisions are stored on the server, 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 *n* to a number less than its current value. |

| Modifier | Description | Comments |
|---|---|---|
| `+w` | File is always writable on client | Not recommended, because Perforce manages the read-write settings on files under its control. |
| `+x` | Execute bit set on client | Used for executable files. |

## Specifying how files are stored in the server

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 server to the 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 client 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.

## 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` flag.

## 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 the client 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 $` |
| `$Change$` | Perforce changelist number under which file was submitted | `$Change: 439 $` |
| `$Date$` | Date of last submission in format *`YYYY/MM/DD`* | `$Date: 2000/08/18 $` |
| `$DateTime$` | Date and time of last submission in format *`YYYY/MM/DD hh:mm:ss`*<br>Date and time are as of the local time on the Perforce server at time of submission. | `$DateTime: 2000/08/18 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 $` |

# Index