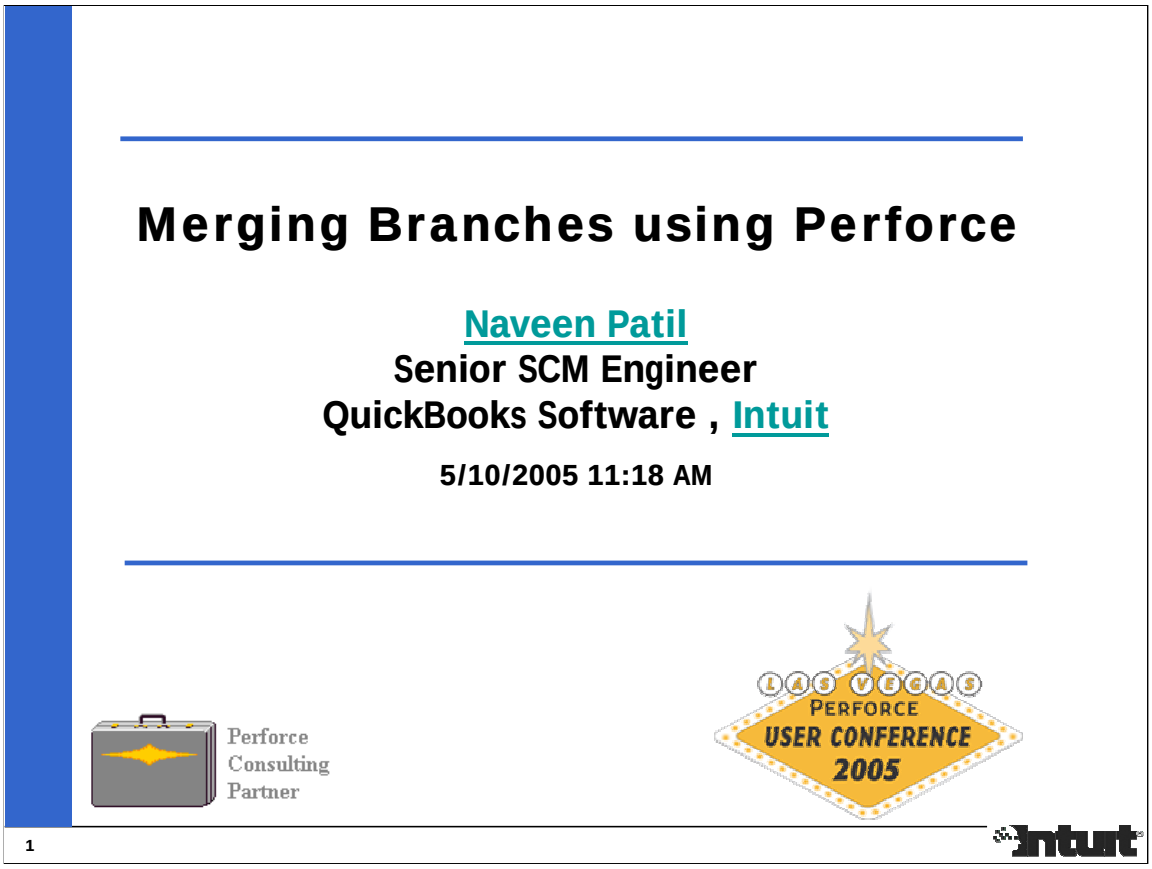


Notes:

Abstract:

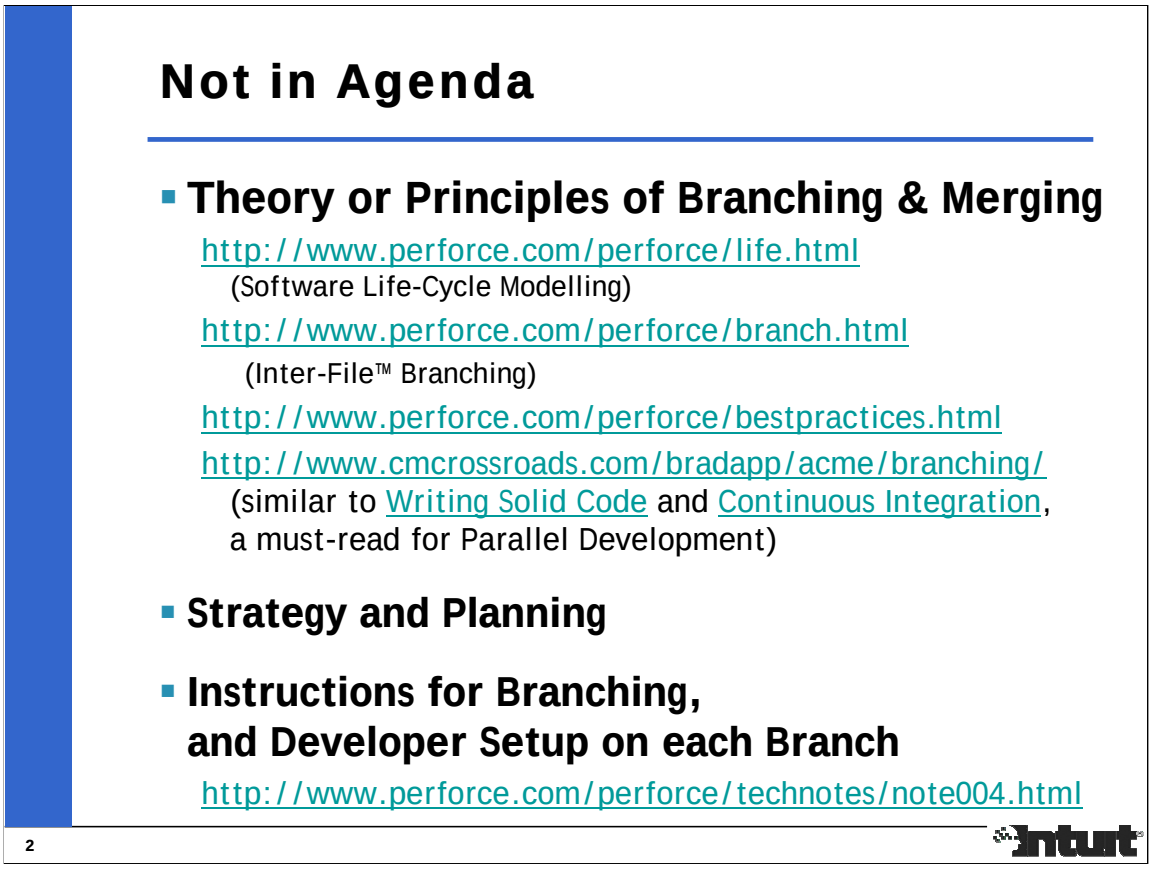
During the past 3 years, the need to do Parallel Development on the QuickBooks product at Intuit has evolved from using a Main branch for development and a Release branch for Change Control to several streams of development that use Main almost only for integration between branches. The associated frequency and cost of merging these branches has also significantly increased.

This presentation focuses on describing the practical learnings from using Perforce for Merging between branches. It discusses the repeatable procedure that has been successfully used to merge between branches, the algorithm Perforce uses to find the Base and Contributor versions, the merge conflict resolution and it's effect on future merges, and developer best practices that can aid this task. Several examples are used to illustrate the decisions associated with resolving a merge conflict. The significant improvements in the 2004.2 release that help with merging are also discussed.

Biography:

Naveen Patil (http://public.perforce.com/guest/naveen_patil/pct.html), Intuit (www.intuit.com)

Naveen works at Intuit as Senior SCM Engineer in the QuickBooks Group. Since Oct 1989, he has worked in various SCM related positions at companies like TiVo, Silicon Graphics, Kubota Graphics and Olivetti. He has also consulted on SCM tasks at companies like Corsair Communications and Perspecta. He is a Perforce Consulting Partner since Mar 2004 and Certified Trainer since Oct 2004. Merging Branches using Perforce is the topic of Naveen's talk.



Notes:

Each product should have a documented Branching and Merging Strategy for Parallel Development. It should be communicated and well understood by all stakeholders - Project Management, Engineering, Quality Control, SCM, etc. It's objectives should include (i) support for development on multiple releases at the same time, (ii) avoid the pitfalls of branching and merging, (iii) reduce the time spent on making decisions, (iv) conform to best practices, and (v) increase the confidence in knowing where to submit Changes and their merge into other branches.

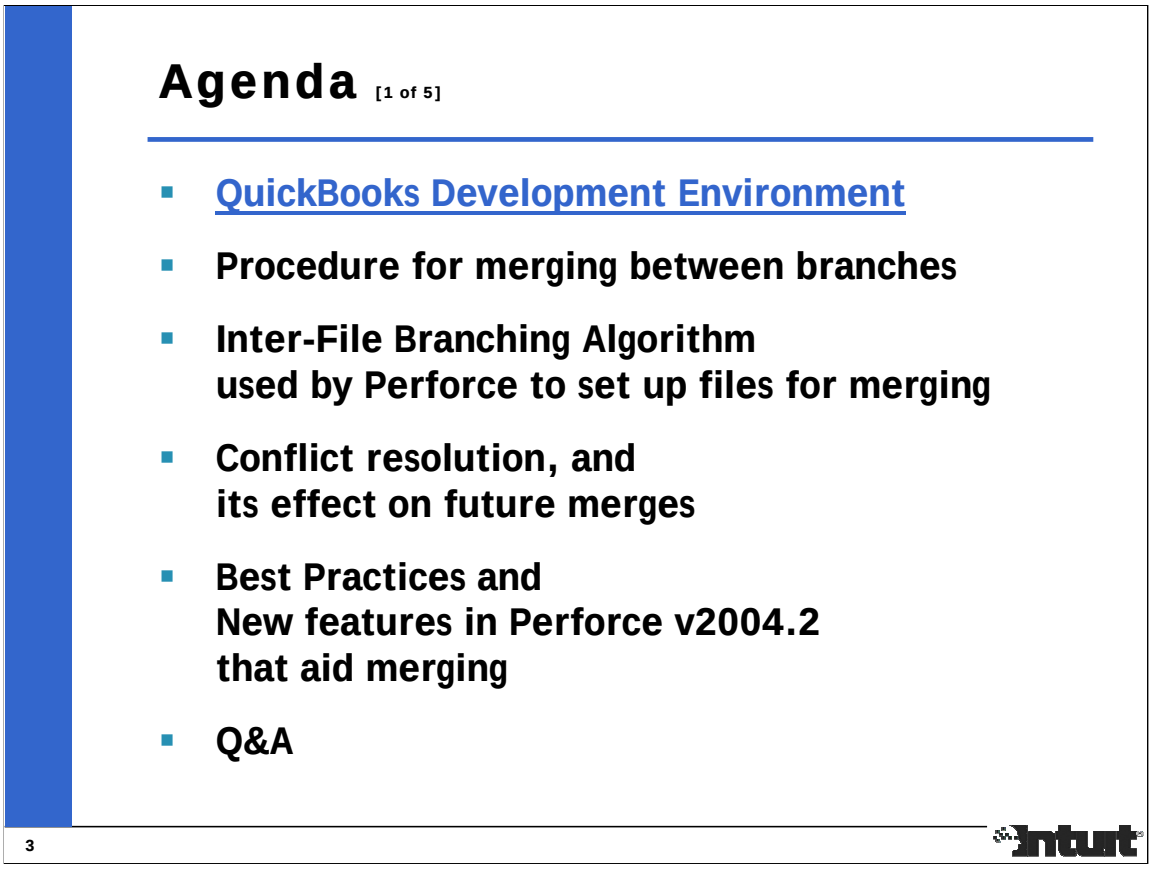
Also See:

Writing Solid Code
(http://c2.com/cgi/wiki?WritingSolidCode)

Continuous Integration
(http://www.martinfowler.com/articles/continuousIntegration.html)

The Importance of Branching Models in SCM
(http://csdl.computer.org/comp/mags/co/2002/09/r9031abs.htm)

Advanced SCM Branching Strategies
(http://www.vance.com/steve/perforce/Branching_Strategies.html)



Notes:

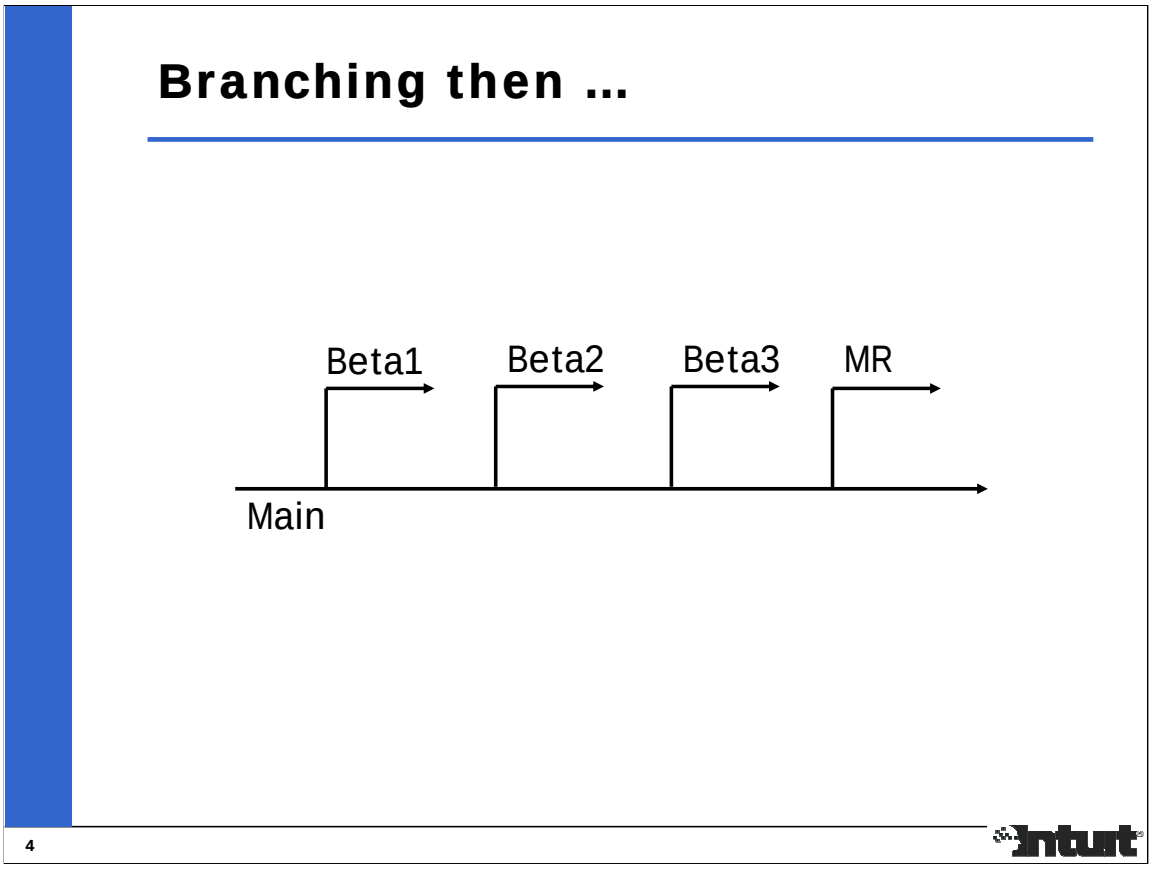
Discuss briefly the evolution of branching needs for QuickBooks development, and the Daily Workflow that promotes Continuous Integration for both Concurrent and Parallel Development.

Next, focus on Merging for Parallel Development, and list the steps that cover the different possibilities that can exist in a merge.

Then discuss the contribution of Perforce to a merge and the manual contribution needed to resolve conflicts. Illustrate the effects of each type of resolution with examples. Perforce contributes some merge errors too, and these get fixed quickly, so the tool is getter better with each release. It is harder to control the errors contributed by humans.

Finally, describe the development best practices and the improvements in Perforce v2004.2 that aid the task of merging branches.

Emphasize that the tool cannot make up for lack of planning or shifting plans. Like any other tool, Perforce can be only as good as humans use it. Explain that there are 2 realities – (i) business needs and (ii) strengths/limitations of Perforce as a tool to support development, and while we cannot let a tool drive business goals, we should also find ways to rein in changes to development plans to make prudent use of Perforce and not tie knots in the integration history that become harder and harder to understand and untangle.

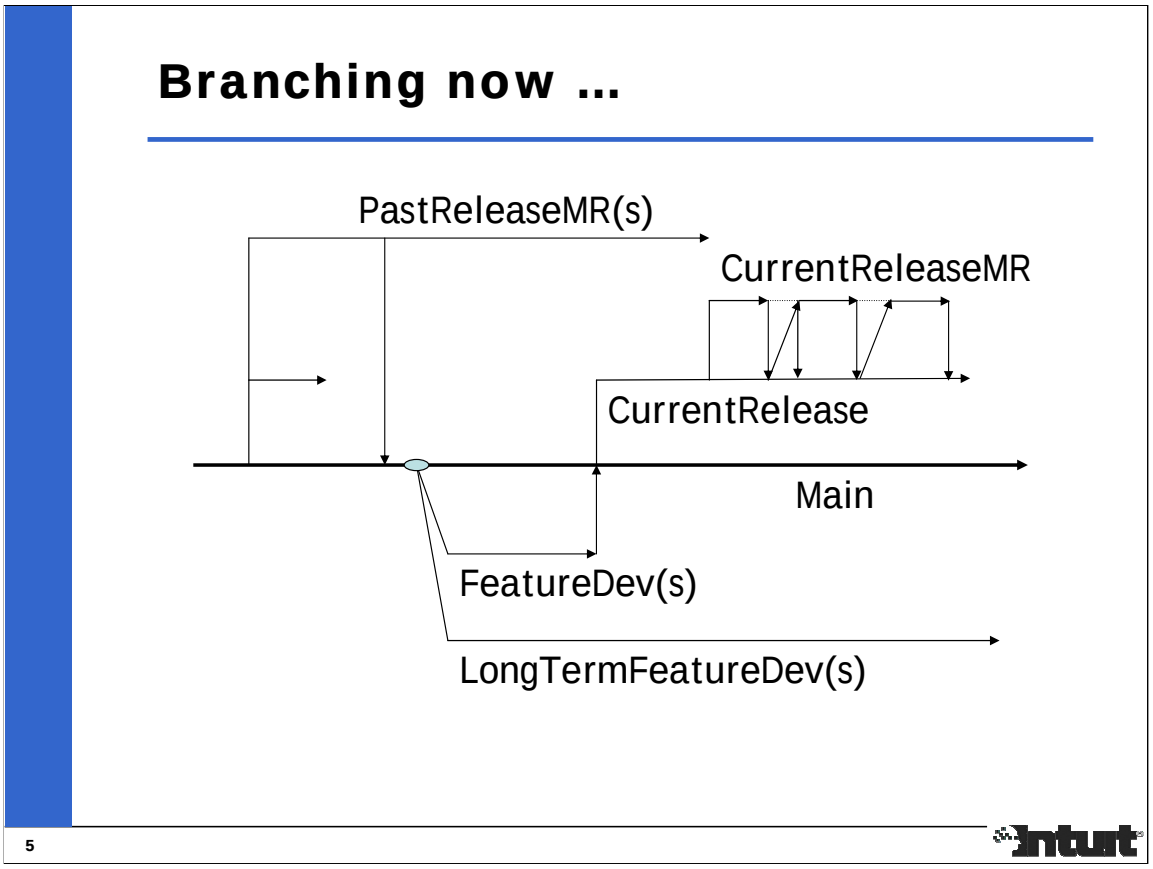


Notes:

Until a few years ago, the release cycle for QuickBooks development consisted of one major release per year followed by several slipstream (maintenance) releases for bugfixes and compliance related changes. A simple branching strategy was used. The Main branch was used for ongoing development of features for the annually scheduled release. To support Change Control for an upcoming milestone, a release branch was forked off of the Main branch. Changes approved on the release branch had to be either checked into both the Main and Release branches (double checkin) or only checked into the Release branch. The release branches were short-lived, and there was no need for merging back from the release branches.

The need to support more than one major release per year, past releases for up to 3 years, and new development, some with a multi-year schedule, lead to development of a branching strategy. Multiple checkins could not be used any more because each branch has it's own policy and schedule for accepting changes.

As we developed code on multiple branches, we had to deal with the more arduous task of merging these branches, and do it in a way that would help these branches converge towards a single code base that included new features as well as bugfixes, and could serve as base for future development.



Notes:

Typically, we start development on a new release from a shipped release.

In early development, we may have several branches (FeatureDev(s) and LongTermFeatureDev(s)) to isolate development on new features.

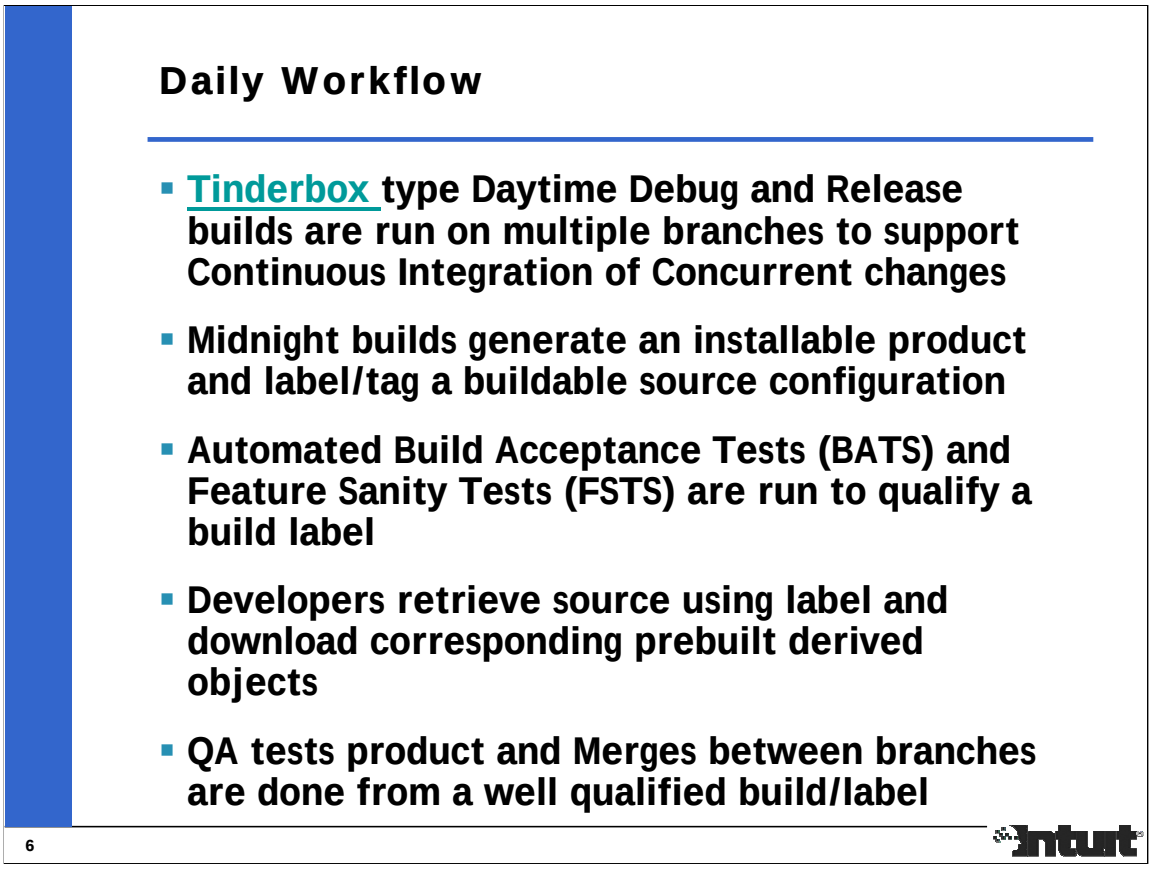
At some point, some of these early development branches (FeatureDev(s)) become stable enough and are approved for integration into the main development branch for the next release. The remaining ones are either deferred to a future release, or are abandoned.

The main development branch for the next release (CurrentRelease) goes through an integration phase of new features and bugfixes from older release, and reaches it's Feature Complete milestone, at which point, it gets branched off into another branch (CurrentReleaseMR) for Change Control.

During the Beta cycles, the CurrentRelease branch is used for ongoing bugfixes and CurrentReleaseMR branch is used for bugfixes approved by Change Control. There are several merges and mergebacks between the 2 branches

After the Beta cycles, the current release gets shipped off of the Change Control branch CurrentReleaseMR, which is then used for minor (slipstream) releases.

Bugfixes made on these branches get merged back into Main in preparation for the next major release.



Notes:

The QuickBooks Development Environment uses the following workflow to support concurrent and parallel development:

Tinderbox type Debug and Release builds are continuously run on multiple branches during the day to provide Continuous Integration of concurrently developed changes. They ensure the branch is in a buildable state after each checkin.

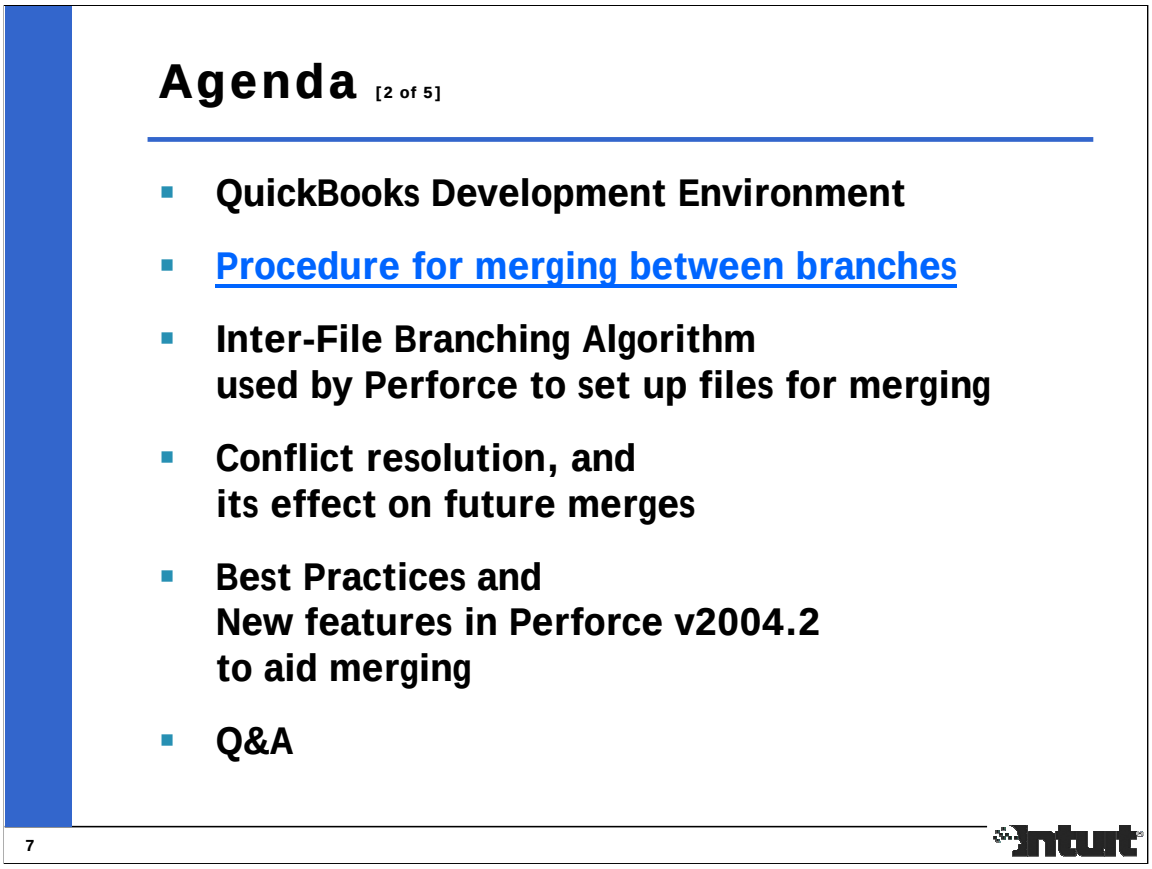
Midnight builds are run off the Changelist built into the last successful daytime build. They generate an installable product and create a label, which is used to identify each build. Automated Build Acceptance Tests (BATS) and Feature Sanity Tests (FSTS) are used to qualify each build label.

Developers retrieve source into their Client using the label created by the midnight build, and download corresponding derived objects. This ensures that they start their daytime work with a buildable product.

QA starts testing a well qualified build that has already passed basic install and feature sanity tests. This ensures the quality of QA testing and makes better use of their time.

This workflow is followed every single day to ensure that the product is in a usable state.

Merges between branches are also done from a well qualified label to ensure the quality of changes being propagated from one branch to another. Builds and Tests are run after each merge to determine the quality of merged code.

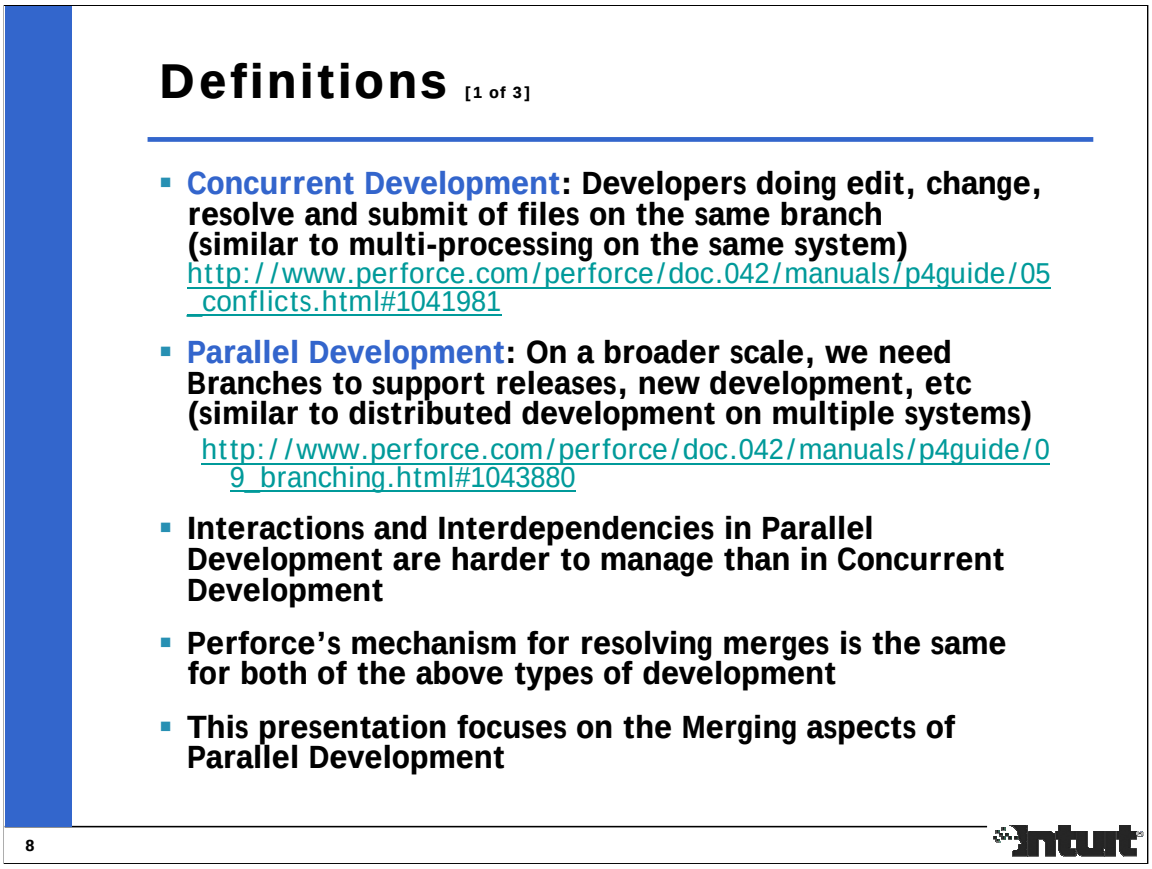


Notes:

The next section of slides discuss the terminology and procedure associated with merging between branches used to support parallel development.

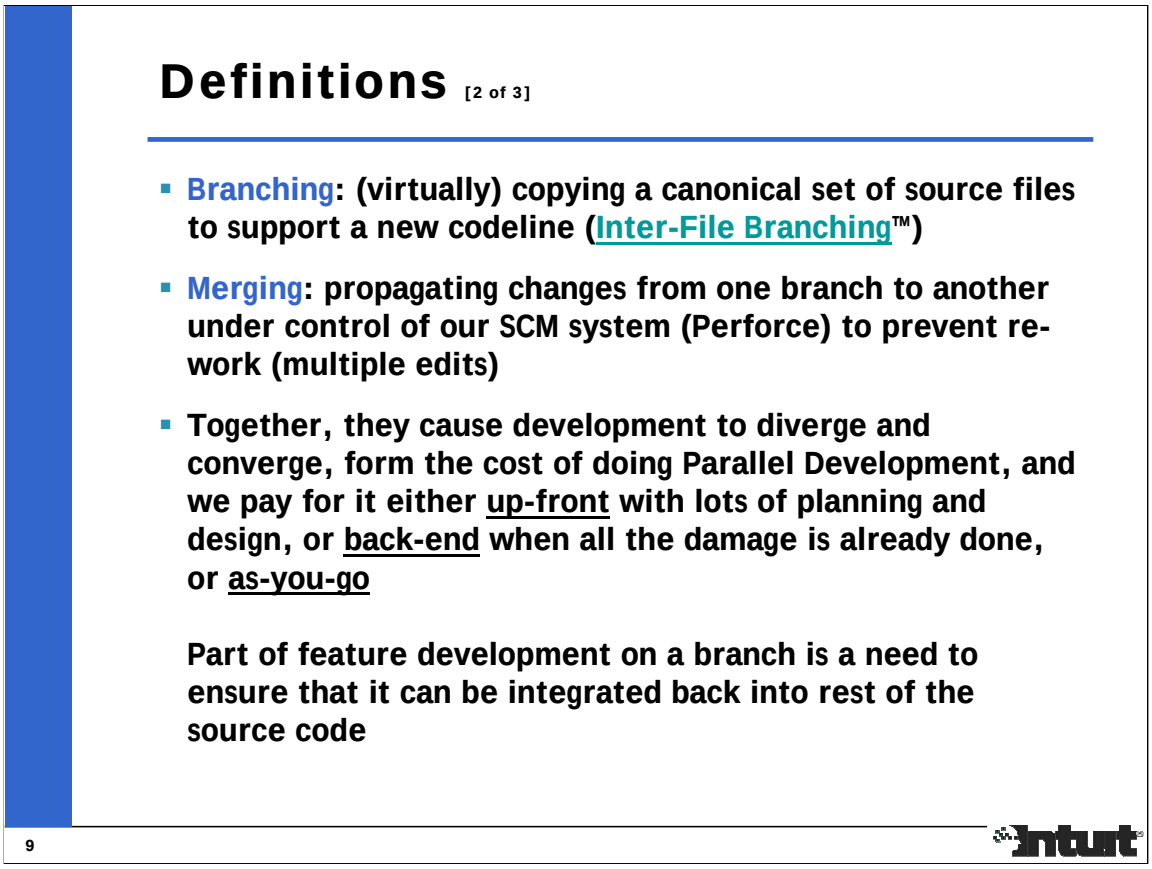
Each step in the procedure may not have files associated with it in every merge, but making it part of the procedure ensures that all the different cases are accounted for. The set of files associated with each case are preferably submitted in independent changelists to aid debugging of merge errors.

Command line names and options have been used in the instructions because they can be concisely presented. But, P4Win can be used too, and “P4 to P4Win Translation Guide” (http://www.perforce.com/perforce/doc.042/manuals/cmd2win/index.html) can be used to map between the CLI (P4) and GUI (P4Win).



Notes:

Just as problems like file locking, synchronization and inter-process communication between processes in a distributed system are harder than that of multi-processing on the same system, interactions and interdependencies in Parallel Development are harder to manage than in Concurrent Development.



Notes:

Concurrent Development leads to the benefits of Continuous Integration as users edit, resolve and submit changes on the same branch. Since developers modifying the same set of files are familiar with the code, they can usually resolve conflicts much easier than merging branches that have diverged for a while. Since the duration of divergence between developers workspaces is usually small, most conflicts are easily resolved.

Similarly, regular planned merges between branches as-you-go helps control the complexity of each merge. In a large scale project, it may not be feasible to merge between branches per Changelist (see http://www.perforce.com/perforce/conf2003/berarducci/berarducci.pdf and http://www.perforce.com/perforce/conf2003/berarducci/berarducci_ppt.pdf) because of differences in policy and release management, so merges need to be planned less frequently. The frequency of merges can be either biweekly, weekly, monthly or at certain stability milestones depending on the type of changes happening on the branches. The responsibility of negotiating and maintaining a Branch/Merge Plan falls into the SCM group.

Code Champions and Architects need to evaluate at design time the impact of developing a new feature on code being developed on other branches.

Setting up the p4review.py review daemon and getting email notification of changes to specific parts of the source tree is an effective way of communicating changes to the source tree and reducing the impact of surprises.

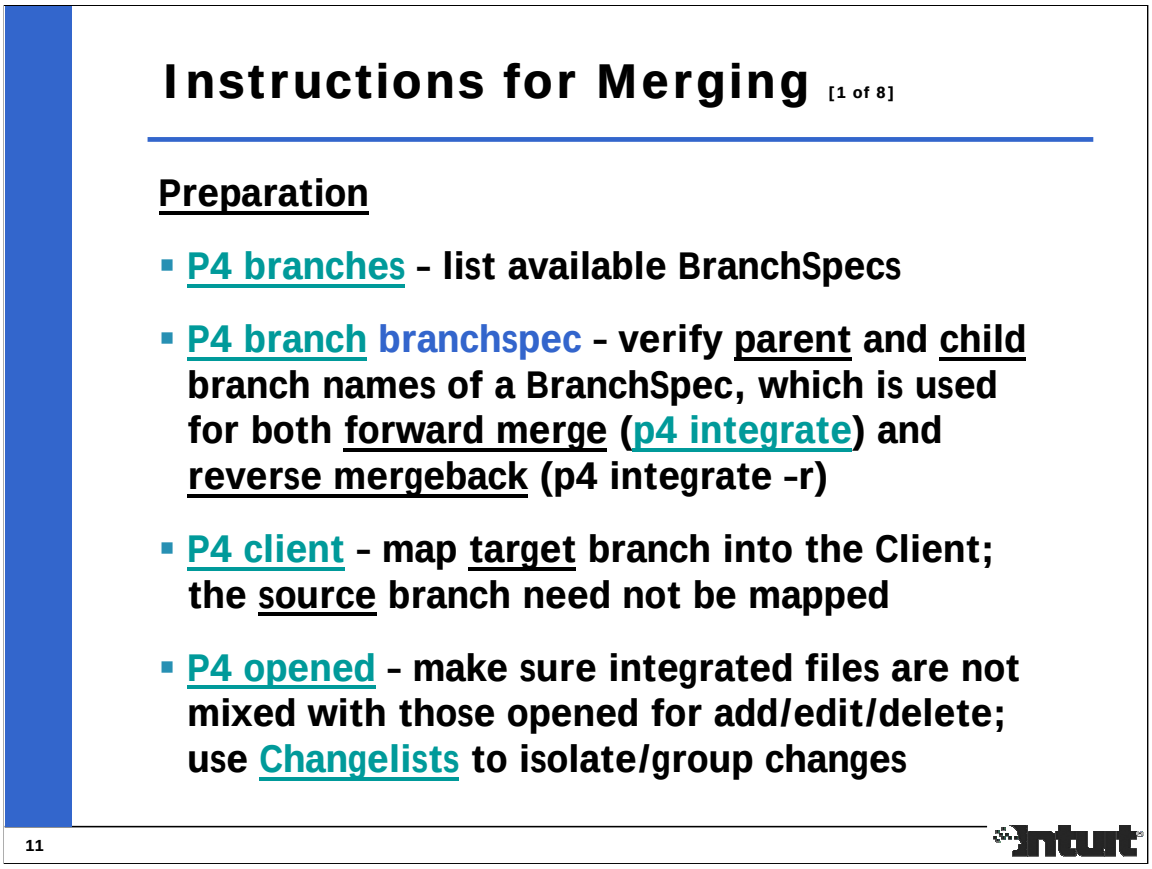
Definitions [3 of 3]

- **Parent and Child Branches**
  - Perforce Branches have an integration hierarchy, as defined by mappings in BranchSpecs
  - Main branch is the Trunk, and normally has no Parent
- **Source and Target Branches**
  - Integrations can happen in either direction
  - "from branch" (THEIRS) is Source, and "into branch" (YOURS) is Target
- **Forward Merge (Rebase) and Reverse Mergeback**
  - Forward Merge has Parent as Source, and Child as Target
  - Reverse Mergeback has Child as Source and Parent as Target
- **Complete, Selective and Subtractive Merge(back)**
  - Including all files until a reference point is Complete
  - Can be Selective using Changelists or Versions, partially too
  - Subtractive Merge(back) is commonly referred to as "rolling back"
- **Direct and Indirect Merge(back)**
  - Branches in a Direct Merge(back) have Parent/Child relationship
  - Branches in an Indirect Merge(back) are related in integration hierarchy, but have no direct Parent/Child relationship

10

intuit

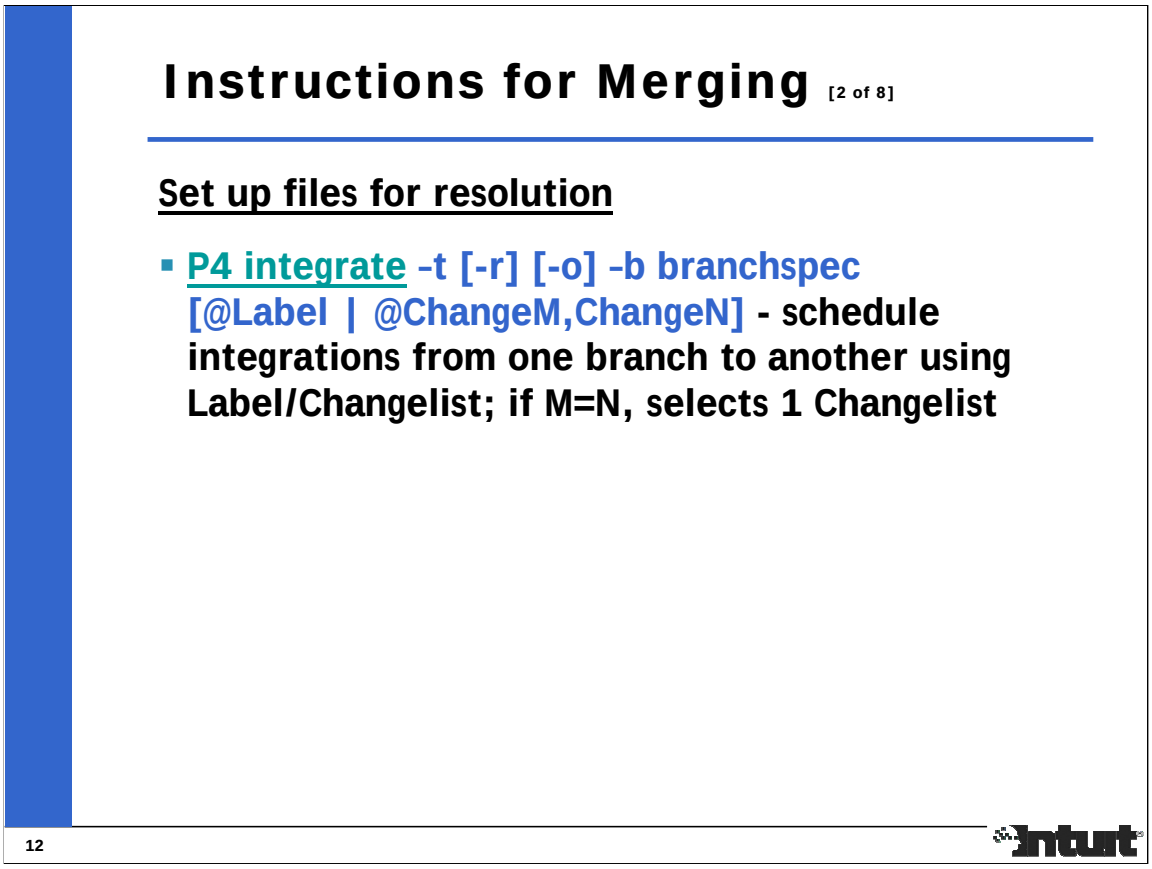
Notes:



Notes:

Branch names are normally same as those of BranchSpecs, and are used interchangeably in different contexts, but they mean different things. A BranchSpec lists mappings between source paths to target paths. A Branch name is normally the target path. Assuming each depot starts off with a Main branch, each new branch created will have a specification and target path named the same, and the “p4 integrate –r” option is used to refer to the reverse mapping.

Perforce supports integration using either a BranchSpec or a FileSpec. Using a BranchSpec has the benefits of (i) being able to list branch specifications using “p4 branches”, (ii) displaying mappings in branch specification using “p4 branch –o <branchspec>”, and (iii) concisely reusing the mappings using “p4 integrate –b <branchspec>”. A FileSpec is normally used to move, rename or split files within a branch.



Notes:

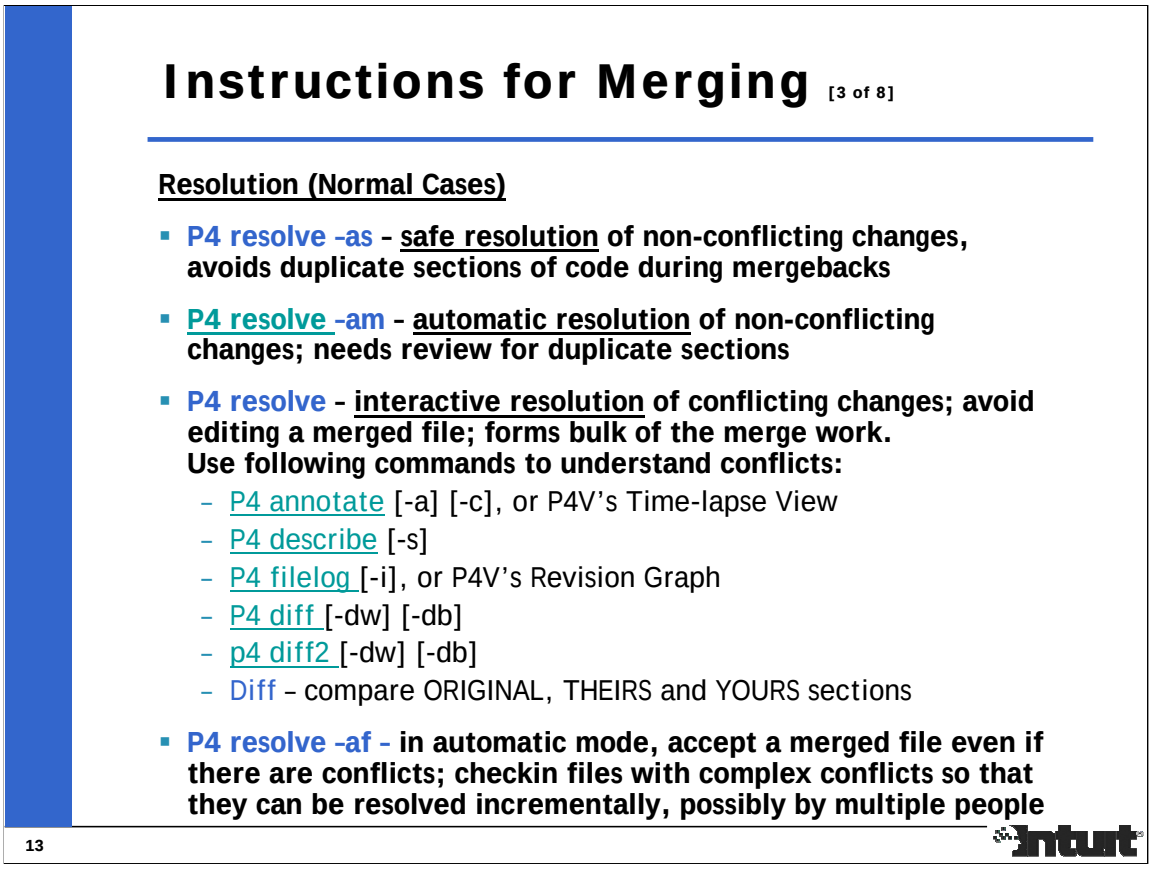
"p4 integrate –b branchspec" is similar to "p4 sync" used in concurrent development to set up an edited file for resolve with newer version(s) checked into the depot.

The "p4 integrate -t" option is needed to propagate filetype changes from one branch to another.

The "p4 integrate –o" option displays the base file name and revision which will be used in subsequent resolves if a resolve is needed.

The "p4 resolve –o" option displays the base file name and revision which will be used during the the merge.

As described in "p4 help undoc", @=ChangeM can be used as an alternative to @ChangeM,ChangeM to selectively integrate a single Changelist.



Notes:

Safe and Automatic resolution ([-as] and [-am]) are reliable and significantly reduce the amount of work required to do the merge. Manual resolution of conflicts have much lower reliability, and require a review process to provide redundancy.

To understand conflicts, it's often easier to save the ORIGINAL, THEIRS and YOURS sections of each conflict into different files and use a tool like Araxis Compare (http://www.araxis.com) to inspect the differences between them, with and without white space changes. "p4 resolve" has options too for listing these differences, and they compare the entire files, not the ORIGINAL, THEIRS and YOURS sections of each conflict .

The [-af] option is useful for resolving files with a large number of conflicts or when conflicts need to be resolved by several people. By using it, we lose the benefits of using resolve options and visual merge tools. If files get checked in with conflict markers, then conflict blocks become part of annotated history.

Note that "p4 resolve –f" uses the saved contents of the previous resolve action as YOURS version, and this is different from the YOURS version of the original resolve, which only has changes from the target branch or edited file.

Conflict resolution forms the foundation of the merge, and affects the overall stability of the merged code base and quality of current and future merges. Lingering "merge errors" are hard to diagnose and reduce confidence in the merge procedure.

Instructions for Merging [4 of 8]

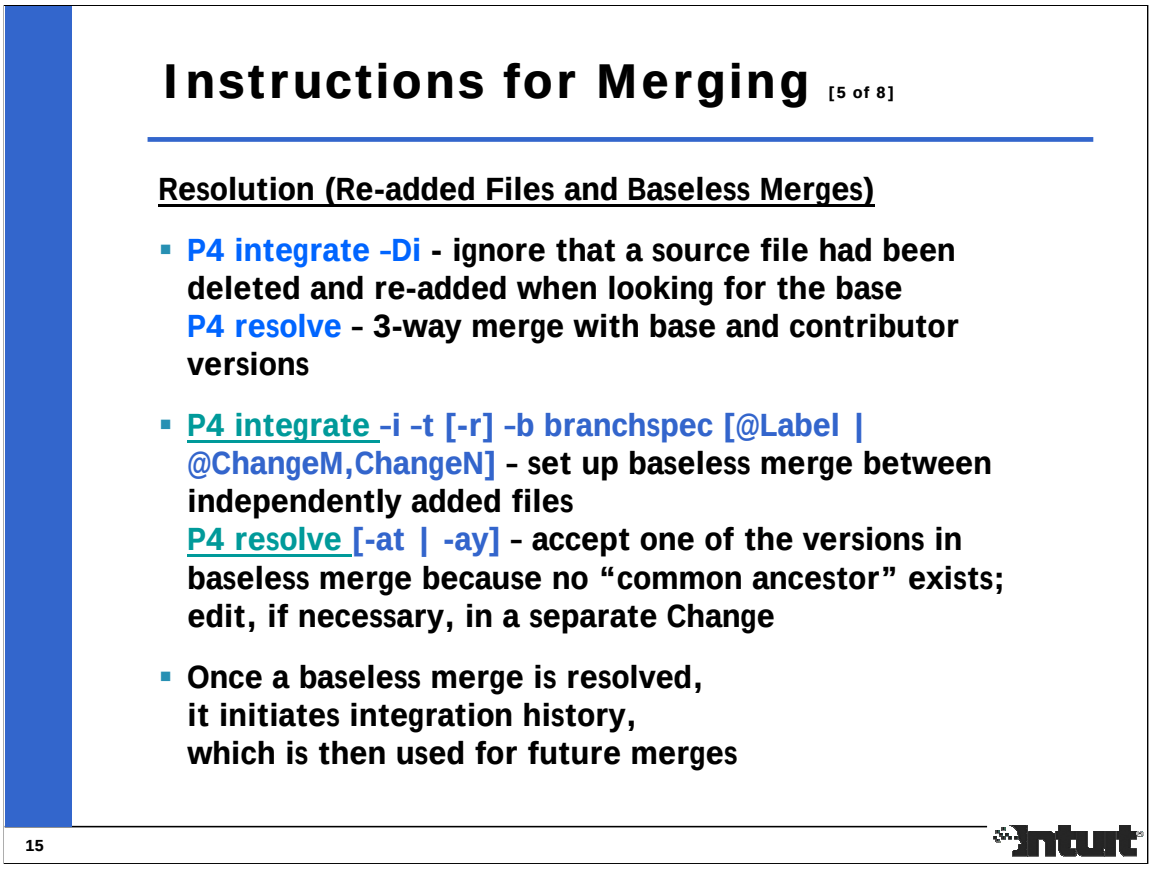
Set up files for resolution (Special Cases)

- **P4 integrate -t [-r] [-o] -b branchspec [@Label | @ChangeM,ChangeN] – repeated command to list files that need [-i] or [-d] option because it's easy to miss this information from the earlier output**

14

intuit

Notes:

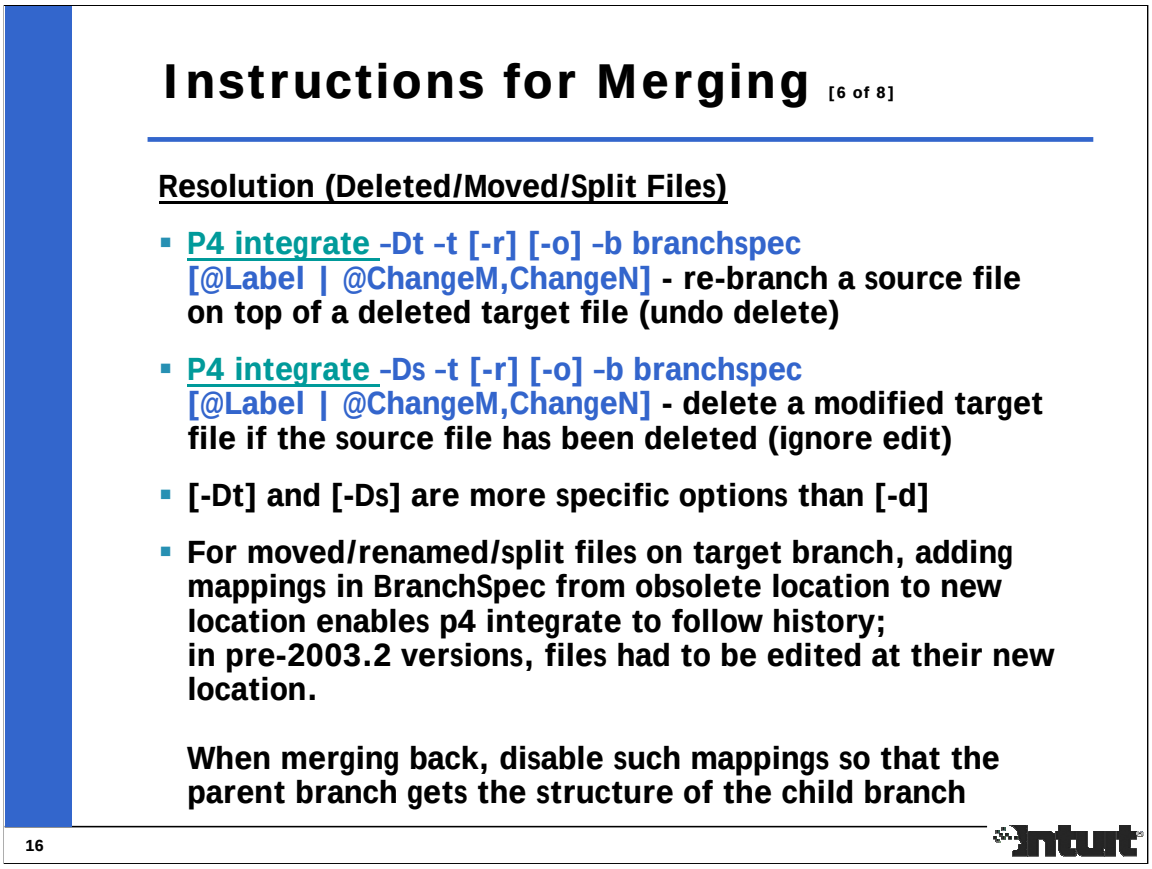


Notes:

If "p4 integrate –i" is used on files that have been re-added, the re-added version will be chosen as the base for the resolve, possibly skipping unintegrated versions before the delete. For re-added files, file history needs to be reviewed before using either [-Di] or [-i].

When the same filenames are independently added on multiple branches, sometimes with different content, Perforce has no integration history between them, so cannot find a common ancestor version. The "p4 integrate –i" option is needed to set up a baseless merge between the files, and the entire contents on one of the contributor versions needs to be accepted; if changes are needed in the merged version, they need to be edited in a separate Changelist to avoid an "add into/from" type of conflict resolution. Once a baseless merge is resolved, it initiates integration history, which is then used for future merges.

NOTE008 How do you get deleted files back?
(http://www.perforce.com/perforce/technotes/note008.html)

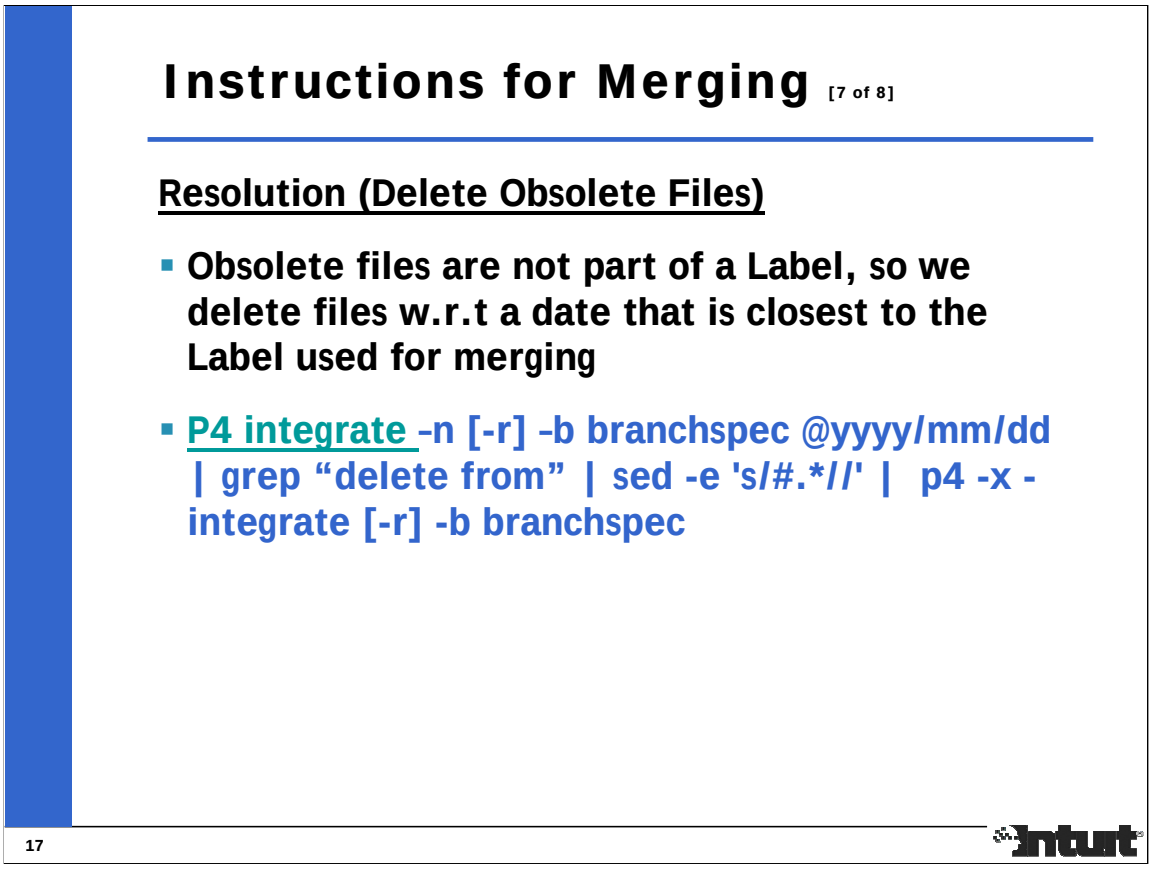


Notes:

SCM Process for handling Moved/Renamed/Split files

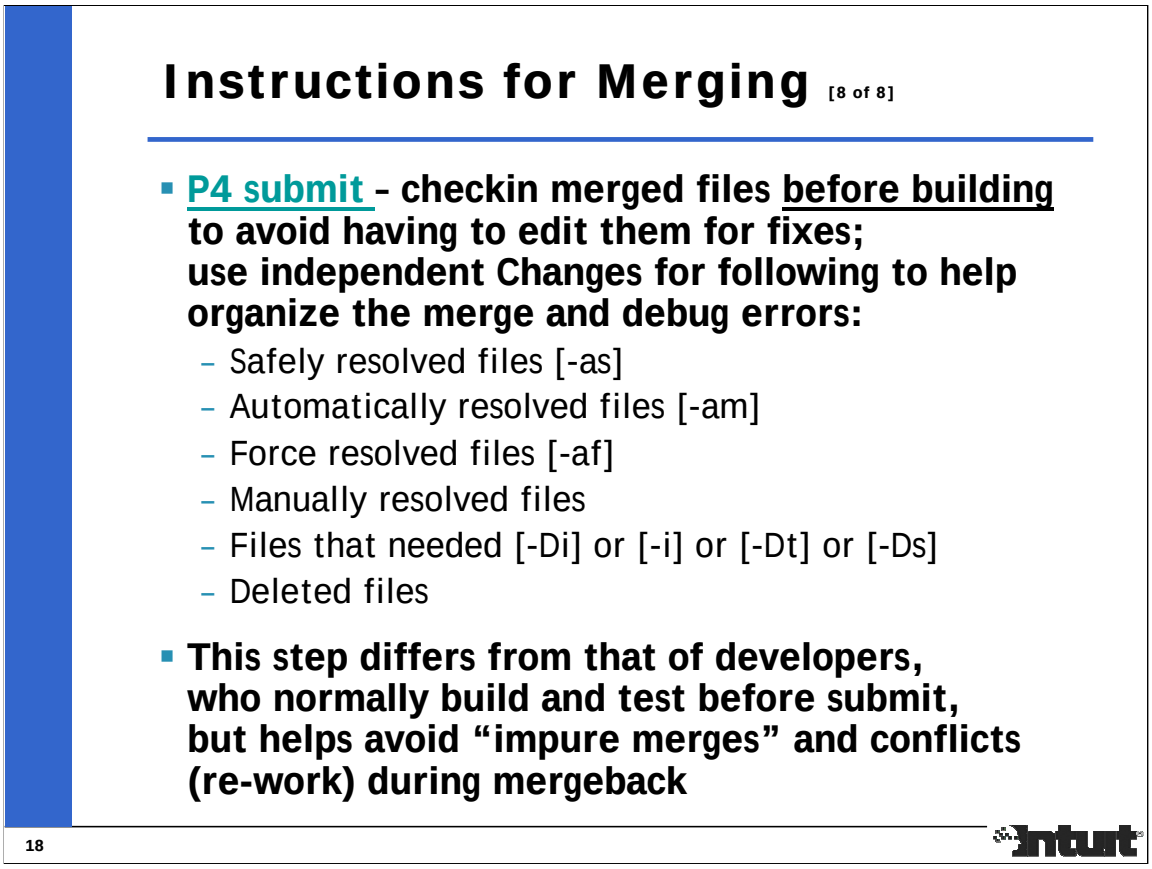
When splitting files, integrate from original file to all destination files.
When moving or renaming files, integrate from original to destination files.
Add mapping in BranchSpec from original location on parent branch to destination location on child branch; note that in the case of split files, the last mapping overrides the earlier ones, which can then be merged using FileSpec.
When merging back, disable such mappings so that the parent branch gets the structure of the child branch.

Perforce maintains integration records when branching/merging and moving/renaming/splitting files, and these can now be followed across multiple integration levels. By adding mappings in a BranchSpec from obsolete location to new location, bugfixes from release branches can now be integrated into the new location of files.



Notes:

“p4 integrate //srcPath/…#delete //destPath/…” can also be used, but may include files that have been deleted after the labels was created. Note that “#delete” is an unsupported revision specifier documented in “p4 help undoc”.



Notes:

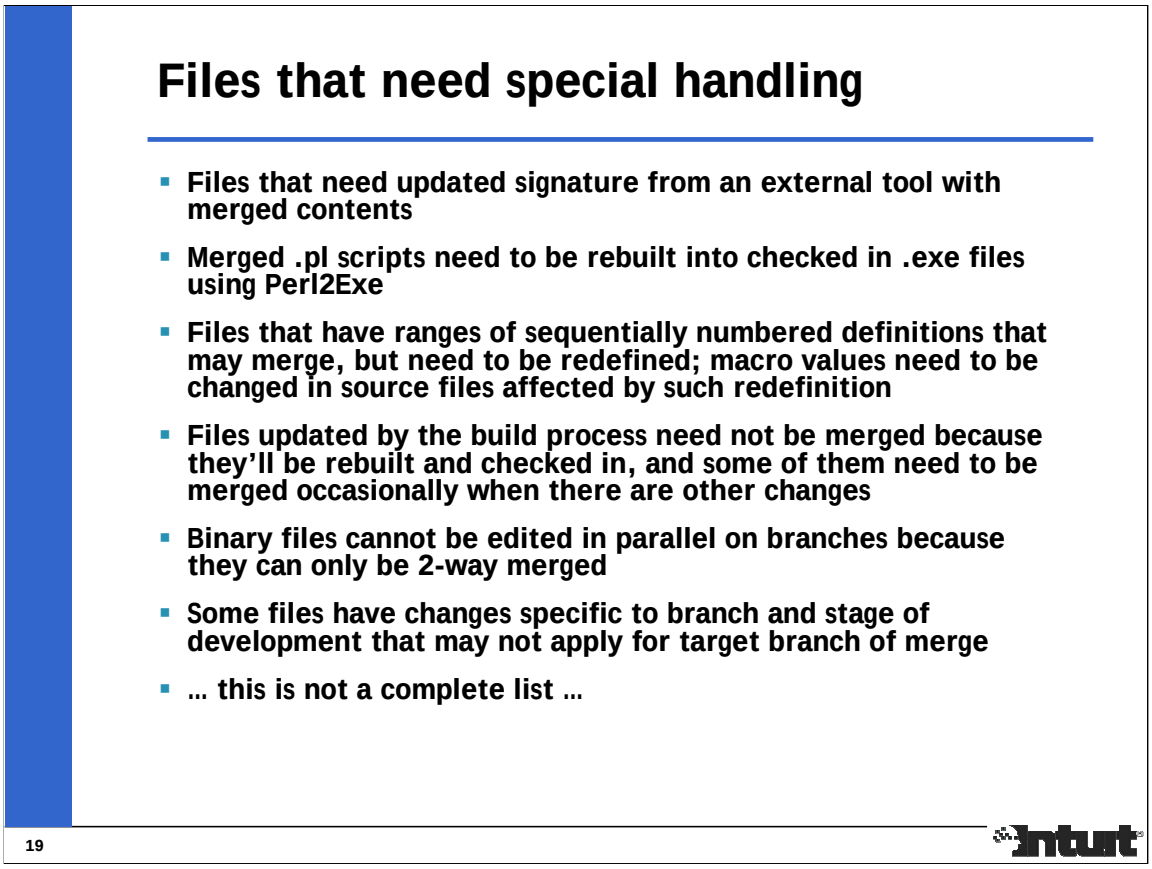
Keep similarly resolved files together in the same Changelist.

As described in http://www.perforce.com/perforce/branch.html (Inter-File Branching), "impure merges" are those versions that had edits in the same Changelist as resolved changes, and they are identified by integration records "edit from/into" or "add from/into".

When a version created by an "impure merge" is merged back into its originating branch, Perforce presents it and the original change as a conflict, which then needs to be re-evaluated. When merging back from the outer level branches, such conflicts form re-work because they're not automatically resolved. They're normally resolved by [-at] (accept theirs) when merging back from development branches.

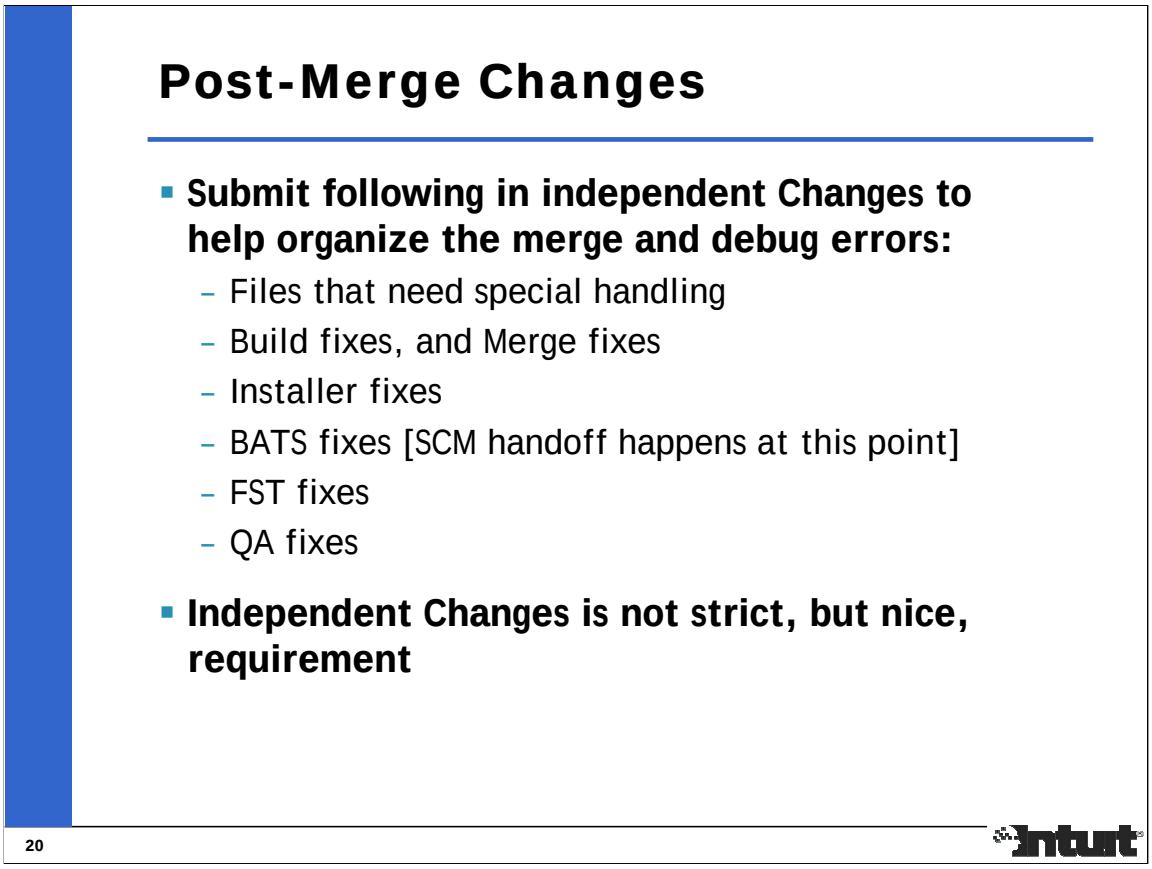
White space introduced in an "impure merge" cause Perforce to accept duplicate sections of code from both contributor versions and split/misaligned YOURS section in conflicts. See CALL #670605 - There are a few known snags with the merge algorithm, most of which involve duplication of code during a mergeback of an edited resolve – empty lines and white space cause different/split Chunks.

To prevent "impure merges", changes with resolved conflicts are checked in even if the resolved file is broken because of changes needed in sections that didn't conflict.



Notes:

This knowledge comes from understanding the source and build infrastructure, and is product specific.



Notes:

Use the merge capabilities of Perforce before applying syntactic/semantic/product knowledge to fix what it doesn't understand.

Resist the temptation to fix all types of errors in the same Change.

Developers often suggest "accept theirs" or "accept yours" to workaround the conflict resolution process so that they can edit their way through a merge, but that's re-work that merging is supposed to avoid and the integration records that get created affect future merges/mergebacks.

"p4 integrate –h –f <revision range>" can be used to force Perforce to re-integrate versions that have already been integrated, resolved and submitted. The [-h] option is needed to make integrate use the same target version on the client (the '#have' revision) as the earlier one. The revision range used by the earlier integrate should be noted from its integration record, and specified for the re-integrate.

Agenda [3 of 5]

- QuickBooks Development Environment
- Procedure for merging between branches
- Inter-File Branching Algorithm used by Perforce to set up files for merging
- Conflict resolution, and its effect on future merges
- Best Practices and New features in Perforce v2004.2 to aid merging
- Q&A

21

intuit

Notes:

Inter-File Branching Algorithm [1 of 2]

http://www.perforce.com/perforce/branch.html **(Section 3)**

http://www.perforce.com/perforce/technotes/note057.html

http://www.perforce.com/perforce/technotes/note065.html

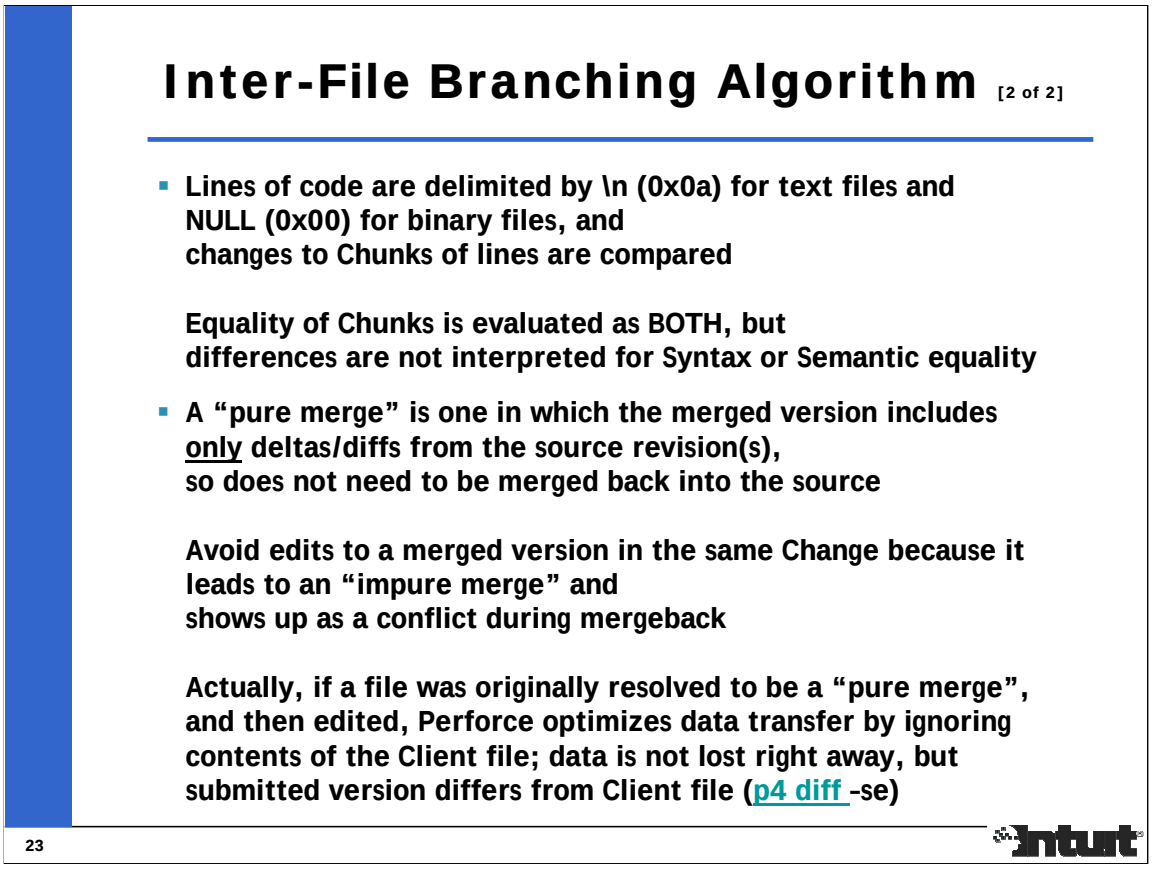
http://www.perforce.com/perforce/doc.042/manuals/p4guide/09_branching.html#1043880

- **This is what Perforce (tool) contributes to the Merge**
- **p4 integrate determines the common ancestor, base (ORIGINAL) and contributor versions (THEIRS/SOURCE and YOURS/TARGET)**
- **Integration records maintain an audit trail of versions that have already merged, and integration credit is given for versions already merged, so subsequent merges are incremental based on previous one(s)**
- **Common Ancestor + Integration Credits = Base Version(s)**

22

intuit

Notes:



Notes:

When resolving conflicts, a good rule of thumb is to limit edits to sections of code within the conflict markers. If such edits completely resolve a file, then it avoids the need to edit them in a separate Changelist. If a purely merged file needs to be edited, then do so in a separate Changelist that is checked in after the one that contains the merge.

With a pure-merged file, Perforce saves the resolved contents in a read-only file to discourage further edits. "p4 edit" needs to be used to make the file writable before edits, and doing so makes the merge impure by changing the resolve action to "accept edit". The general rule of thumb is that if it's read-only in the workspace, edits made to the file after changing its attributes won't be transferred to the server during submit. "p4 resolved" displays the resolve action on files that haven't yet been submitted into Perforce.

Agenda [4 of 5]

- QuickBooks Development Environment
- Procedure for merging between branches
- Inter-File Branching Algorithm
  used by Perforce to set up files for merging
- Conflict resolution, and
  its effect on future merges
- Best Practices and
  New features in Perforce v2004.2
  that aid merging
- Q&A

24

intuit

Notes:

Conflict Resolution

- Conflicts occur when comparison of the 2 contributor versions to the base yields differences in matching chunks. Resolution is the task of reviewing the differences and fixing them.
- This is what the integrator (human) contributes to the Merge
- Resolve decisions affect future merges and mergebacks. A special advantage is gained by doing “pure merges” which need not be merged back (helps avoid future work)
- Irrespective of resolve decision, source revision(s) always get integration credit for future merges, so subsequent merges are incremental w.r.t all previous ones; hence, pay as-you-go
- Each conflict is individually resolved, but there is no tracking of how individual conflicts within a file were resolved
- For “pure merges”, the resulting diff after merge should show changes that came from the other branch
- Isolating and Diagnosing merge errors is hard, especially when conflicts were manually resolved

25

intuit

Notes:

Conflict resolution is different from reviewing the effective differences that exist in a merged version.

There is no easy way of reviewing conflict resolution because conflicting sections of code cannot be differentiated from the non-conflicting ones after the conflicts are resolved and conflict markers removed.

Examples [1 of 15]

```
137  >>>> ORIGINAL Makefile#4
138      $(UTILITIES_LIB_FILES)\
139  ==== THEIRS Makefile#8
140      $(UTILITIES_LIB_FILES)\
141      $(CONTEXTMANAGEMENT_LIB_FILES) \
142      QBGDataSecurity.LIB \
143  ==== YOURS Makefile
144      $(UTILITIES_LIB_FILES) \
145      $(CONTEXTMANAGEMENT_LIB_FILES) \
146  <<<<
```

26

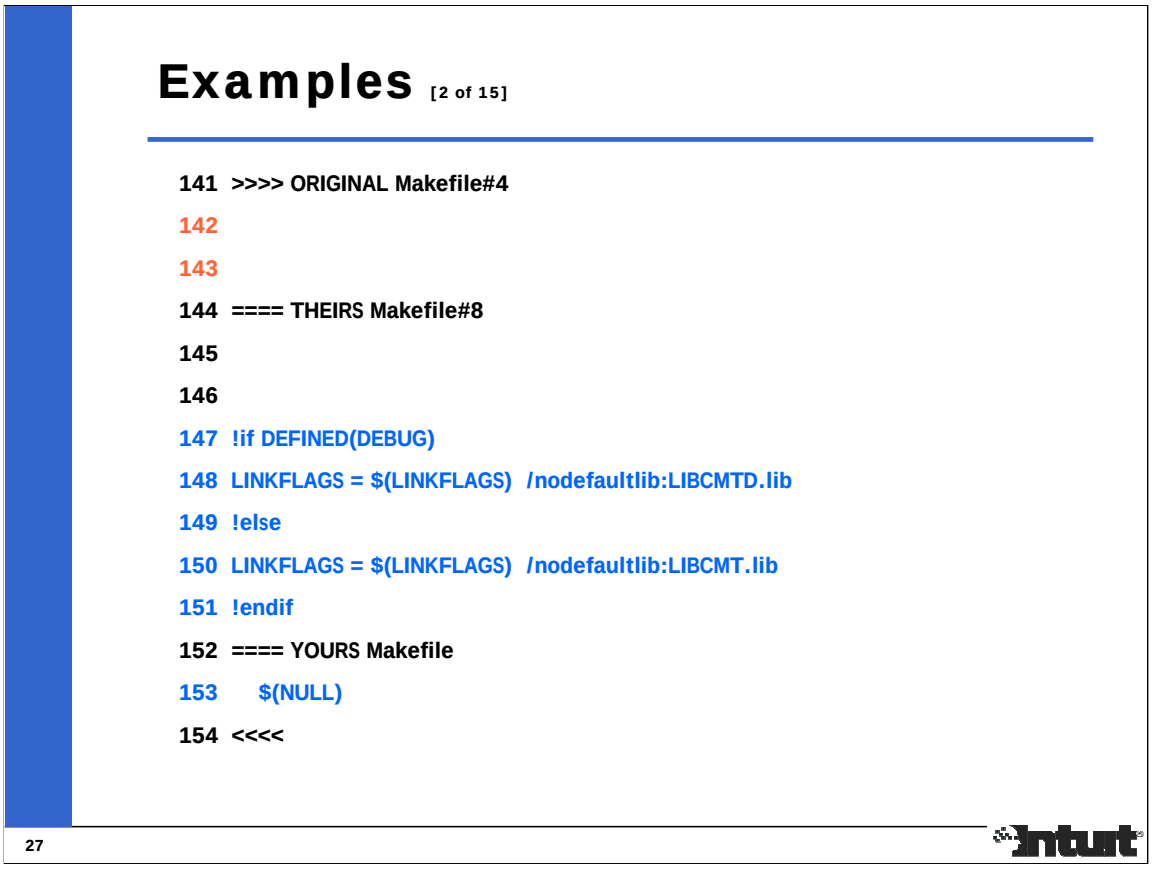
Notes:

The ORIGINAL and THEIRS sections refer to contents in the base (Makefile#4) and contributor (Makefile#8) versions on the source branch. They are always displayed above the YOURS section, which refers to content in the contributor version, normally the head revision, on the target branch. “p4 integrate –o” and “p4 resolve –o” can be used to display the base file name and revision.

This conflict is resolved by selecting the THEIRS section because it subsumes the non-whitespace changes made in the YOURS section, and inserting a space before the continuation character on the 1st line.

Resolved Result:

```
137    $(UTILITIES_LIB_FILES) \
138    $(CONTEXTMANAGEMENT_LIB_FILES) \
139    QBGDataSecurity.LIB \
```



Notes:

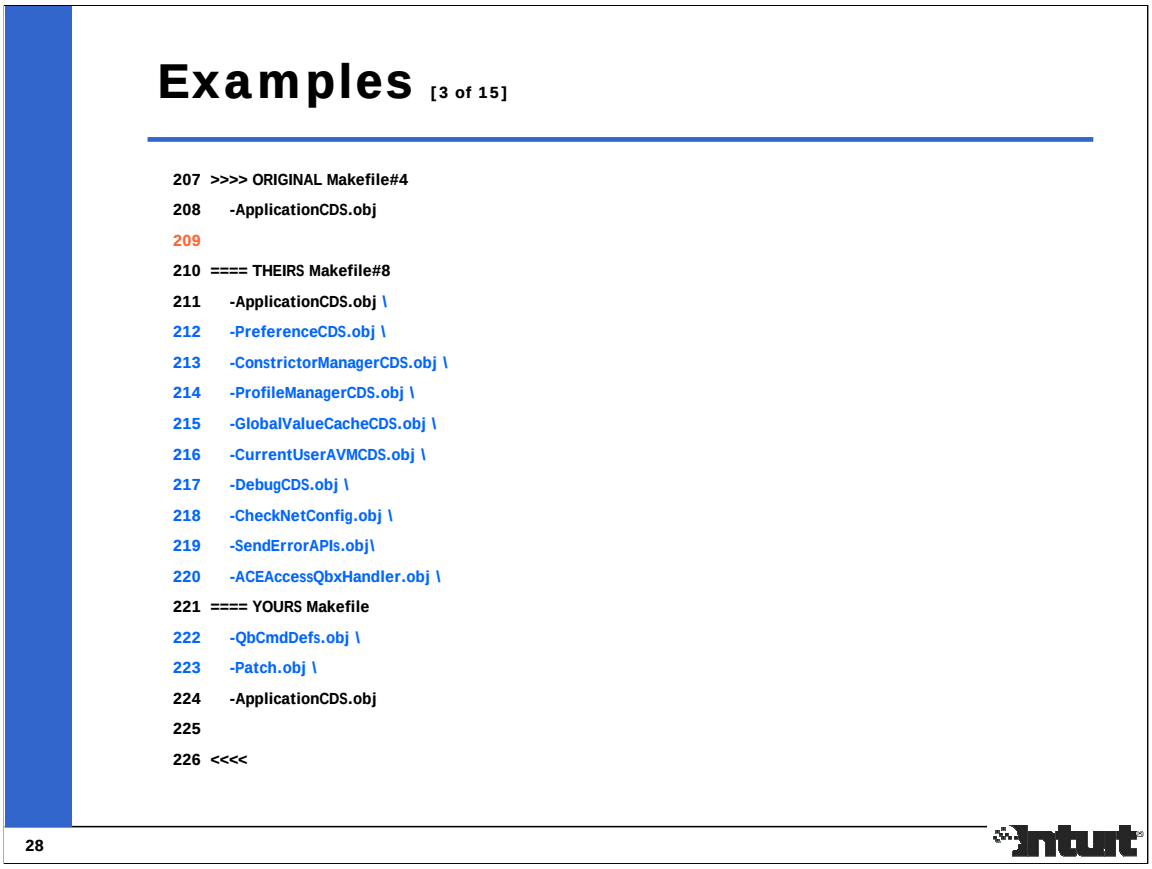
This example has the same base and contributor versions as in the previous one. The change in the YOURS section is actually a continuation of the resolved contents of the previous example.

This conflict is resolved by accepting the contents of both the THEIRS and YOURS sections, but moving the YOURS content higher in the file such that it is immediately after the resolved contents of the previous example.

If an editor is used to display conflicts, then the THEIRS and YOURS sections can be selected by deleting the conflict markers and further edited to resolve them. Some visual merge tools only allow selection of either the THEIRS or YOURS section to resolve a conflict, and in such cases, the merged code needs to be manually edited to add code from the section that wasn't selected.

Resolved Result:

```
140     $(NULL)
141
142 !if DEFINED(DEBUG)
143 LINKFLAGS = $(LINKFLAGS)  /nodefaultlib:LIBCMTD.lib
144 !else
145 LINKFLAGS = $(LINKFLAGS)  /nodefaultlib:LIBCMT.lib
146 !endif
```



Notes:

In this case, additional filenames have been added in both the THEIRS and YOURS sections with a continuation character after ApplicationCDS.obj only in the THEIRS section.

This conflict is resolved by retaining the THEIRS section and appending part of the YOURS section. The continuation character after Patch.obj in the YOURS section can be retained without affecting the interpretation by $(MAKE).

Resolved Result:

```
207    -ApplicationCDS.obj \
208    -PreferenceCDS.obj \
209    -ConstrictorManagerCDS.obj \
210    -ProfileManagerCDS.obj \
211    -GlobalValueCacheCDS.obj \
212    -CurrentUserAVMCDS.obj \
213    -DebugCDS.obj \
214    -CheckNetConfig.obj \
215    -SendErrorAPIs.obj\
216    -ACEAccessQbxHandler.obj \
217    -QbCmdDefs.obj \
218    -Patch.obj \
```

Examples [4 of 15]

```
540  >>>> ORIGINAL CWelcomeDialog.cpp#17
541              QBString encodedFilePath = filePath;
542              encodedFilePath.HexEncode();
543  ==== THEIRS CWelcomeDialog.cpp#20
544              Util::String encodedFilePath = filePath;
545              Util::EncodeURIComponent(encodedFilePath);
546  ==== YOURS CWelcomeDialog.cpp
547              QBString encodedFilePath = filePath;
548              if (GetFullPathName(encodedFilePath.c_str(), MAX_PATH, fullpath, &lpName))
549              {
550                 encodedFilePath = fullpath;
551              }
552              encodedFilePath.HexEncode();
553  <<<<
```

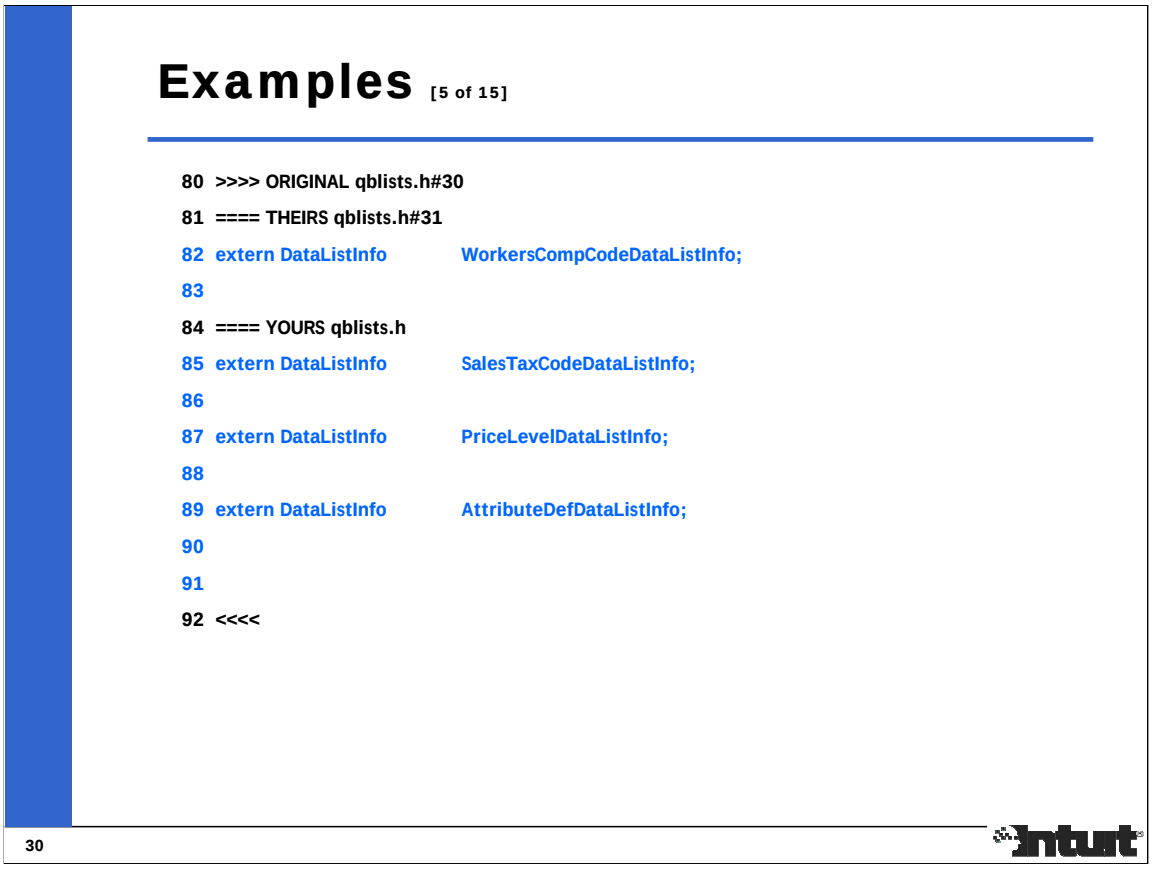
29

Notes:

This conflict is resolved by accepting the THEIRS section with the changes in the YOURS section inserted between the 2 lines.

Resolved Result:

```
540          Util::String encodedFilePath = filePath;
541          if (GetFullPathName(encodedFilePath.c_str(), MAX_PATH, fullpath, &lpName))
542          {
543             encodedFilePath = fullpath;
544          }
545          Util::EncodeURIComponent(encodedFilePath);
```



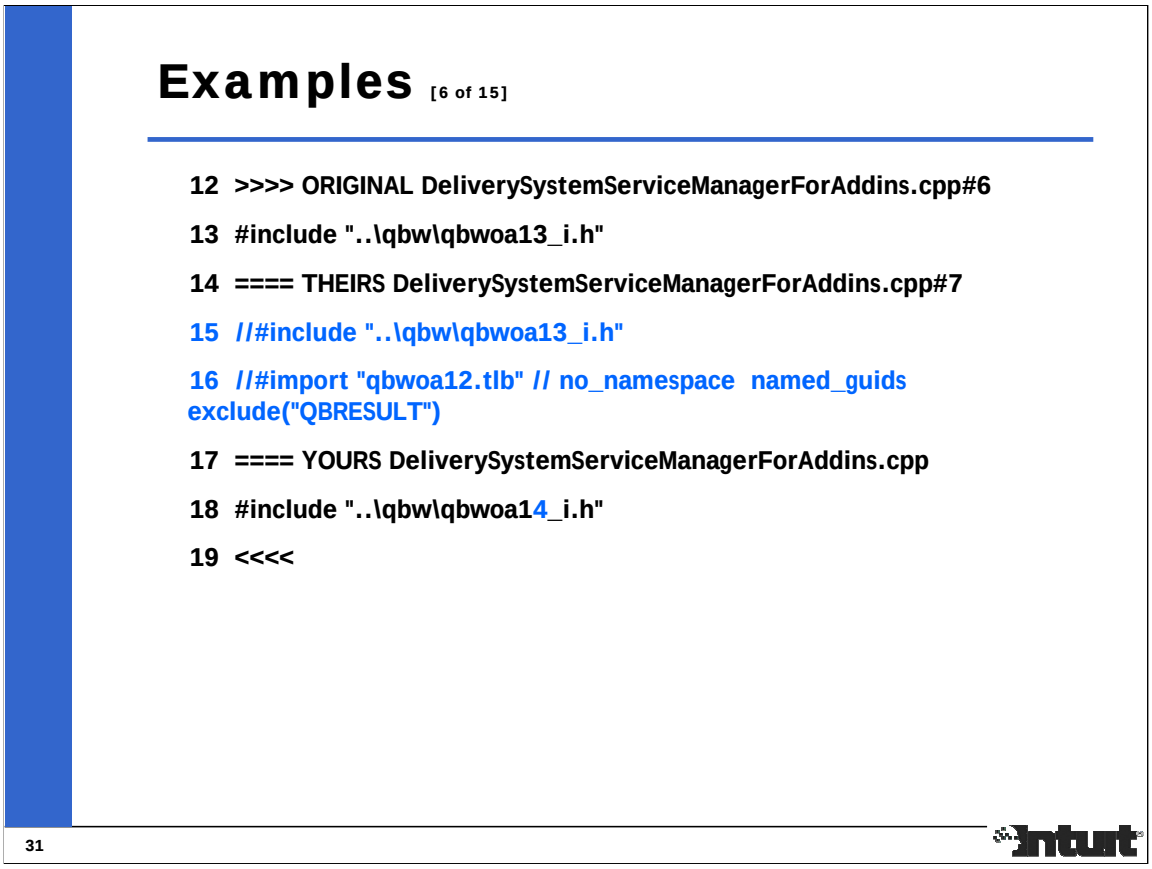
Notes:

A null ORIGINAL section indicates that changes in the THEIRS and YOURS sections were added at the same location.

In this case, variables corresponding to different features were added on the source and target branches, and the conflict is resolved by accepting both sets of features.

Resolved Result:

```
80 extern DataListInfo        WorkersCompCodeDataListInfo;
81
82 extern DataListInfo        SalesTaxCodeDataListInfo;
83
84 extern DataListInfo        PriceLevelDataListInfo;
85
86 extern DataListInfo        AttributeDefDataListInfo;
87
```



Notes:

In this case, the commented lines in the THEIRS section will not affect the compilation of this source file. The filename change from ..\qbw\qbwoa13_i.h to ..\qbw\qbwoa14_i.h is applied in the comment to ensure this file will compile if the comment characters are removed in a later change.

Resolved Result:

```
12  //#include "..\qbw\qbwoa14_i.h“
13  //#import "qbwoa12.tlb" // no_namespace  named_guids exclude("QBRESULT")
```

Examples [7 of 15]

```
1232  >>>> ORIGINAL errids.h#254
1233  #if !defined(NO_NSCP)
1234      #define W_NSCP_DISABLE_REMINDER     671
1235      #define W_WEBWRAPNOTFOUND             672
1236      #define W_WEBWRAPNOMEM               673
1237      #define     W_WEBWRAPERR               674
1238  #endif // !defined(NO_NSCP)
1239  ==== THEIRS errids.h#263
1240  #define W_NSCP_DISABLE_REMINDER     671
1241  #define W_WEBWRAPNOTFOUND             672
1242  #define W_WEBWRAPNOMEM               673
1243  #define     W_WEBWRAPERR               674
1244  ==== YOURS errids.h
1245      #define W_NSCP_DISABLE_REMINDER     671
1246      #define W_WEBWRAPNOTFOUND             672
1247      #define W_WEBWRAPNOMEM               673
1248      #define     W_WEBWRAPERR               674
1249  <<<<
```

32

Notes:

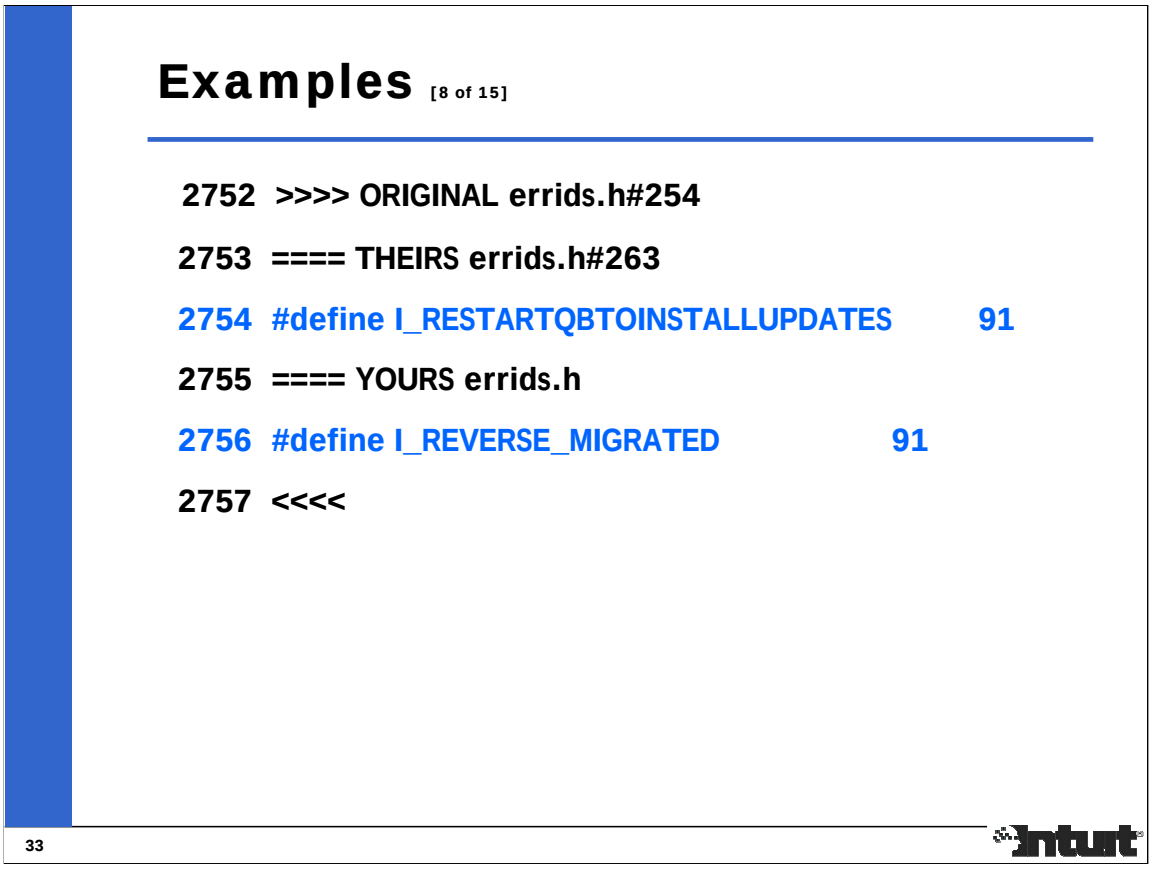
In this case, only whitespace differences exist between the THEIRS and YOURS sections.

The [-db] and [-dw] options of “p4 resolve” can be used to ignore whitespace changes or all whitespace when merging files, and use text from the client file (YOURS), but it may not always be possible to use these options for all conflicts because valid whitespace changes may also exist.

This conflict is resolved y accepting the THEIRS section because it’s indentation matches those of the lines surrounding this conflict.

Resolved Result:

```
1232  #define W_NSCP_DISABLE_REMINDER     671
1233  #define W_WEBWRAPNOTFOUND             672
1234  #define W_WEBWRAPNOMEM               673
1235  #define     W_WEBWRAPERR               674
```



Notes:

This example is similar to Example #5 in having a null ORIGINAL block followed by changes in the THEIRS and YOURS sections added at the same location, but resolving by only accepting both sets of changes will cause a redefinition error from the compiler.

This conflict is resolved by first accepting both sets of changes, and then fixing the redefinition error in an independent Changelist.

Resolved Result:

```
2752  #define I_RESTARTQBTOINSTALLUPDATES      91
2753  #define I_REVERSE_MIGRATED               91
```

Edited Result:

```
2752  #define I_RESTARTQBTOINSTALLUPDATES      91
2753  #define I_REVERSE_MIGRATED               92
```

Examples [9 of 15]

```
14 >>>> ORIGINAL load.c#130
15 #ifdef _QBONLINE_
16 ==== THEIRS load.c#138
17 #include "Util/Platform.h"
18 #include "Util/StringEncoding.h"
19 #include "Util/Chars.h"
20
21 #ifdef _QBONLINE_
22 ==== YOURS load.c
23 <<<<
```

34

intuit

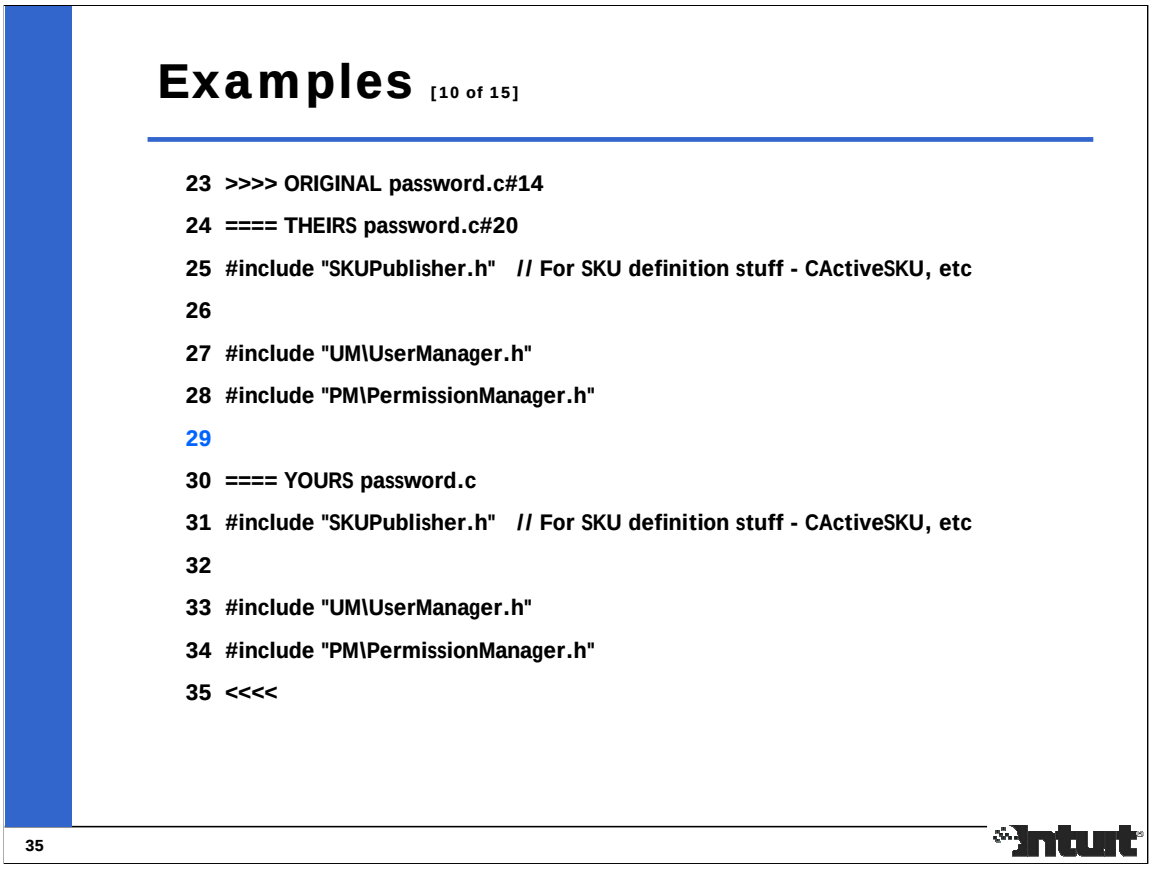
Notes:

In this case 3 filenames and an empty line got added in the THEIRS section, and the contents of the ORIGINAL section got deleted in the YOURS section.

This conflict is resolved by accepting the THEIRS section and applying the deletion from the YOURS section.

Resolved Result:

```
14 #include "Util/Platform.h“
15 #include "Util/StringEncoding.h“
16 #include "Util/Chars.h“
17
```



Notes:

In this case, the empty line #29 is the only difference between the THEIRS and YOURS sections, and such differences are usually caused by users editing changes into multiple branches (double checkins) instead of editing into one branch and merging it into other branches.

This conflict can be resolved by choosing either the THEIRS or YOURS section depending on the text surrounding this conflict.

Resolved Result:

```
23 #include "SKUPublisher.h"   // For SKU definition stuff - CActiveSKU, etc
24
25 #include "UM\UserManager.h“
26 #include "PM\PermissionManager.h"
```

Examples [11 of 15]

```
>>>> ORIGINAL EDLItemAdapter.cpp#1
   pData->salesTaxCode.SetRecNum(static_cast<ELItemPtrType>(element.GetSalesTaxCodeId()));
   pData->paymentMethod.SetRecNum(static_cast<ELItemPtrType>(element.GetPaymentMethodId()));
   pData->prefVendor.SetRecNum(static_cast<ELItemPtrType>(element.GetPreferredVendorId()));
   pData->taxAgency.SetRecNum(static_cast<ELItemPtrType>(element.GetTaxAgencyId()));

   pData->wasImported = static_cast<Boolean>(element.GetWasImported());

==== THEIRS EDLItemAdapter.cpp#4
   pData->salesTaxCode.SetRecNum(static_cast<ListElementIDType>(element.GetSalesTaxCodeId()));
   pData->paymentMethod.SetRecNum(static_cast<ListElementIDType>(element.GetPaymentMethodId()));
   pData->prefVendor.SetRecNum(static_cast<ListElementIDType>(element.GetPreferredVendorId()));
   pData->taxAgency.SetRecNum(static_cast<ListElementIDType>(element.GetTaxAgencyId()));

   pData->wasImported = static_cast<Boolean>(element.GetWasImported());

==== YOURS EDLItemAdapter.cpp
<<<<
```

36

Notes:

In this case, the changes to the type of cast in the THEIRS section are to function calls that have been deleted in the YOURS section, so they can be ignored.

This conflict is resolved by deleting the entire conflict block.

Examples [12 of 15]

```
>>>> ORIGINAL el_citem.c#1
   this->ReadFromDisk();
   m_pItem->m_item.delCount = delCount;
   if (this->SyncMemory()) {
      (ItemList.hList.item)[m_recNum].hdr.bits.delCount = delCount;
   }
   this->SetDirty (USER_DIRTY, HEADER_DIRTY);

   return S_OK;
}     /* CItem::SetDelCount */
==== THEIRS el_citem.c#14
==== YOURS el_citem.c
   this->ReadFromDisk();
   m_pItem->m_item.delCount = delCount;
   if (this->SyncMemory()) {
      (ItemList.hList.item)[m_recNum].hdr.bits.delCount = delCount;
   }
   this->SetDirty (USER_DIRTY, HEADER_DIRTY);
   m_pItem->m_item.userFieldsUpdated = TRUE;

   return S_OK;
}     /* CItem::SetDelCount */
<<<<
```

37

Notes:

In this case, the change in the YOURS section is to a section of code that has been deleted in the THEIRS section, so it can be ignored.

This conflict is resolved by deleting the entire conflict block.

Examples [13 of 15]

```
>>>> ORIGINAL edlistui.c#1

     short runTimeFieldId = pDict-
  >GetRunTimeFieldId(component, fieldId, TRUE);

     DictEntryPtr pDictEntry = pDict-
  >GetDictEntry(runTimeFieldId);

==== THEIRS edlistui.c#21

     ListElementCountType runTimeFieldId = pDict-
  >GetRunTimeFieldId(component, fieldId, TRUE);

     DictEntryPtr pDictEntry = pDict-
  >GetDictEntry(runTimeFieldId);

==== YOURS edlistui.c

     pSortBag->SetFreeSort(freeSort);

     pDictEntry = pDict->GetDictEntry(component, fieldId);

<<<<
```

38

Notes:

In this case, the change made in the THEIRS section is not applicable to the YOURS content because the variable “runTimeFieldId” and it’s initialization are not needed any more. What really happened here is the variable “runTimeFieldId” got renamed to “fieldId”, and the declaration/initialization of “fieldId” got moved up in the file; p4 automatically accepted this earlier initialization as a non-conflicting change. It is possible that the type of “fieldId” needs to be changed to reflect the change in the THEIRS section, but such edits outside of a conflict block are done in a separate Changelist from the one which only resolves conflicts to avoid an “accept edit” type of conflict resolution.

This conflict is resolved by accepting the YOURS section.

Resolved Result:

```
pSortBag->SetFreeSort(freeSort);
pDictEntry = pDict->GetDictEntry(component, fieldId);
```

Examples [14 of 15]

```
>>>> ORIGINAL EDLMemorizedReportAdapter.cpp#1
      pAbsElement->SetRecordNumber(static_cast<ELItemPtrType>(currRecId));
==== THEIRS EDLMemorizedReportAdapter.cpp#4
      pAbsElement->SetRecordNumber(static_cast<ListElementIDType>(currRecId));
==== YOURS EDLMemorizedReportAdapter.cpp
<<<<


      //we want to reuse this element w/o calling SetRecNum because SetRecNum will
   delete the refdata,
      //forcing us to recreate it... a costly sequence when loading a ton of  elements.
   So just clear it
      //and set new recnum manually. jthomas 10/28/04
      pAbsElement->ClearElementForReUse(static_cast<ELItemPtrType>(currRecId))
;
```

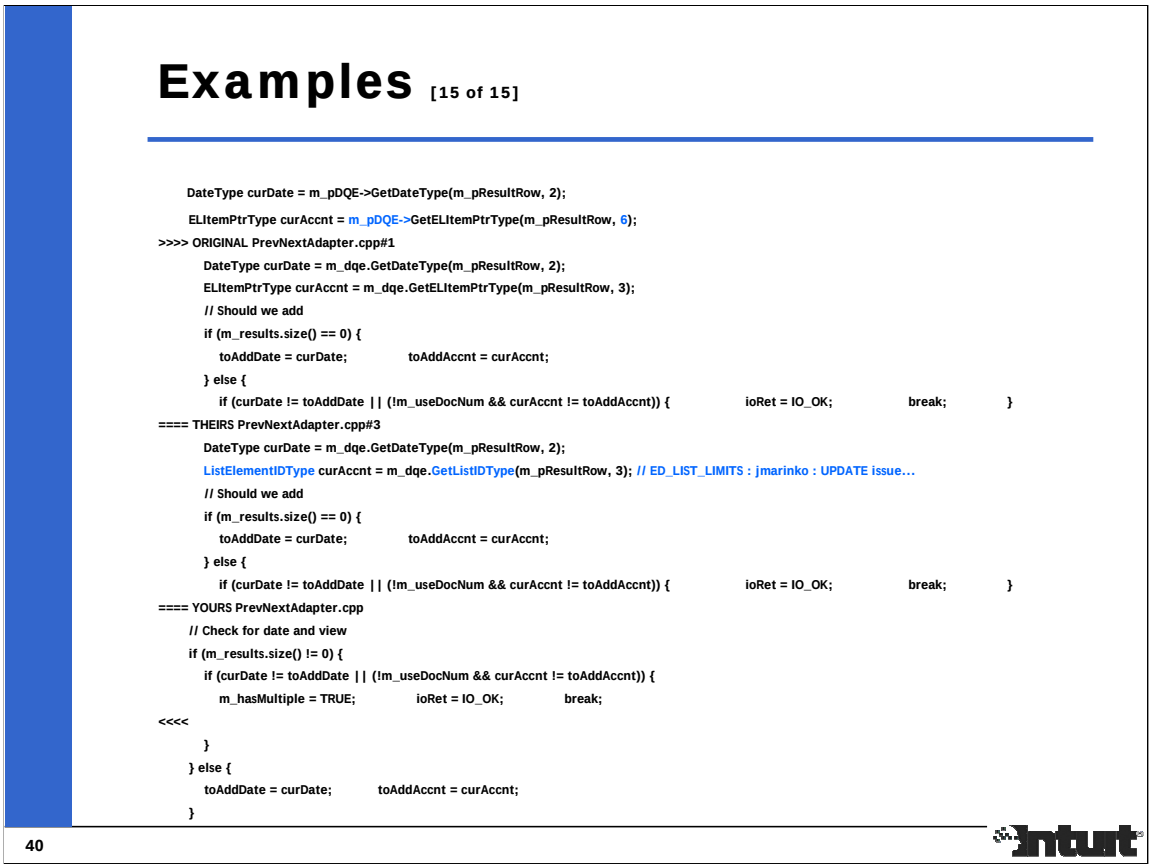
39

Notes:

In this case, the type of cast changed in the THEIRS section, and the empty YOURS section seems to indicate that the function call got deleted, so that change in THEIRS is not needed any more. But, the comment below the conflict block indicates that use of SetRecordNumber() got replaced by ClearElementForReUse(), and it now needs the type of cast change from the THEIRS section.

In such cases, it's hard to decide if edits outside of a conflict should be made in the same Change as the one that resolves conflicts so that related changes are kept together in the same Changelist, or done in a separate Changelist to avoid an "accept edit" type of conflict resolution. Changelists are "units of work" and are often used to selectively merge into other branches, so there is value in keeping related changes together.

This conflict is resolved by deleting the entire conflict block, and applying the type of cast change from the THEIRS block to the section of code below it



Notes:

When there have been large changes on the source and/or target branches, the conflict blocks presented by Perforce often appear out of context with the surrounding sections of code in the target file. The ORIGINAL and THEIRS sections appear similar because they both come from the source branch, and they widely differ from the YOURS section. In such cases, there are 3 possibilities – (i) retain the YOURS section and ignore the changes in THEIRS because they are in code that has been deleted in the target file, or (ii) the changes in THEIRS are in a section that has moved elsewhere in the target file, so the YOURS section is retained in place and the changes in THEIRS are applied to the new location of the code in the target file, or (iii) the changes in THEIRS are in a section that has been moved to another file, so the YOURS section is retained in place and the changes in THEIRS are applied to the new location of the code in the other file, which has to be opened for edit.

In this case, some lines belonging to the target version appear outside the conflict block, and the change in the THEIRS section needs to be applied to a line appearing before the conflict block, which itself has a change. When conflicts become this complex, it is less error prone to make the edits outside the conflict block in the same Changelist as the one that resolves conflicts than making them in separate Changelists to avoid an “accept edit” type of conflict resolution.

Resolved Result:

```
DateType curDate = m_pDQE->GetDateType(m_pResultRow, 2);
ListElementIDType curAccnt = m_pDQE->GetListIDType (m_pResultRow, 6); // ED_LIST_LIMITS :
jmarinko : UPDATE issue...

// Check for date and view
      if (m_results.size() != 0) {
         if (curDate != toAddDate || (!m_useDocNum && curAccnt != toAddAccnt)) {
            m_hasMultiple = TRUE;          ioRet = IO_OK;          break;
                              }
      } else {
         toAddDate = curDate;          toAddAccnt = curAccnt;
```

Resolution (Merge) Tools

http://www.perforce.com/perforce/products/merge.html

http://www.perforce.com/perforce/technotes/note047.html

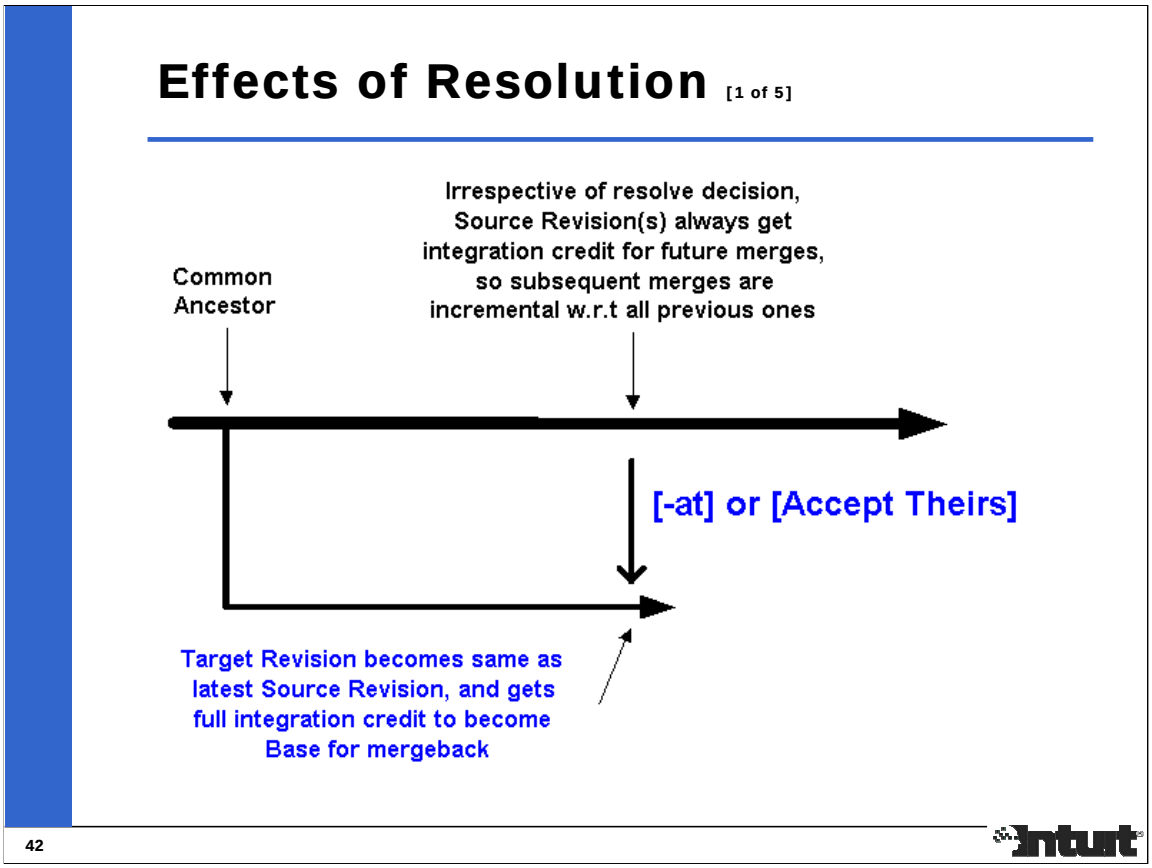
- **Text mode – p4 resolve, provides more control**
- **Visual mode – P4WinMerge and Araxis Merge, provides extensive color coded information**
- **Each mode works better in some cases, and end result is the same assuming the same resolve decisions were made.**

  Perforce compares the merged file with contributor versions to determine the integration credits for the target branch
- **Use what your eyes and fingers like**
- **NOTE047 Using third-party merge tools with Perforce**
- **Some applications may have a way of comparing and merging their proprietary non-text file format, but such files are normally resolved as either [-at] or [-ay]**

41

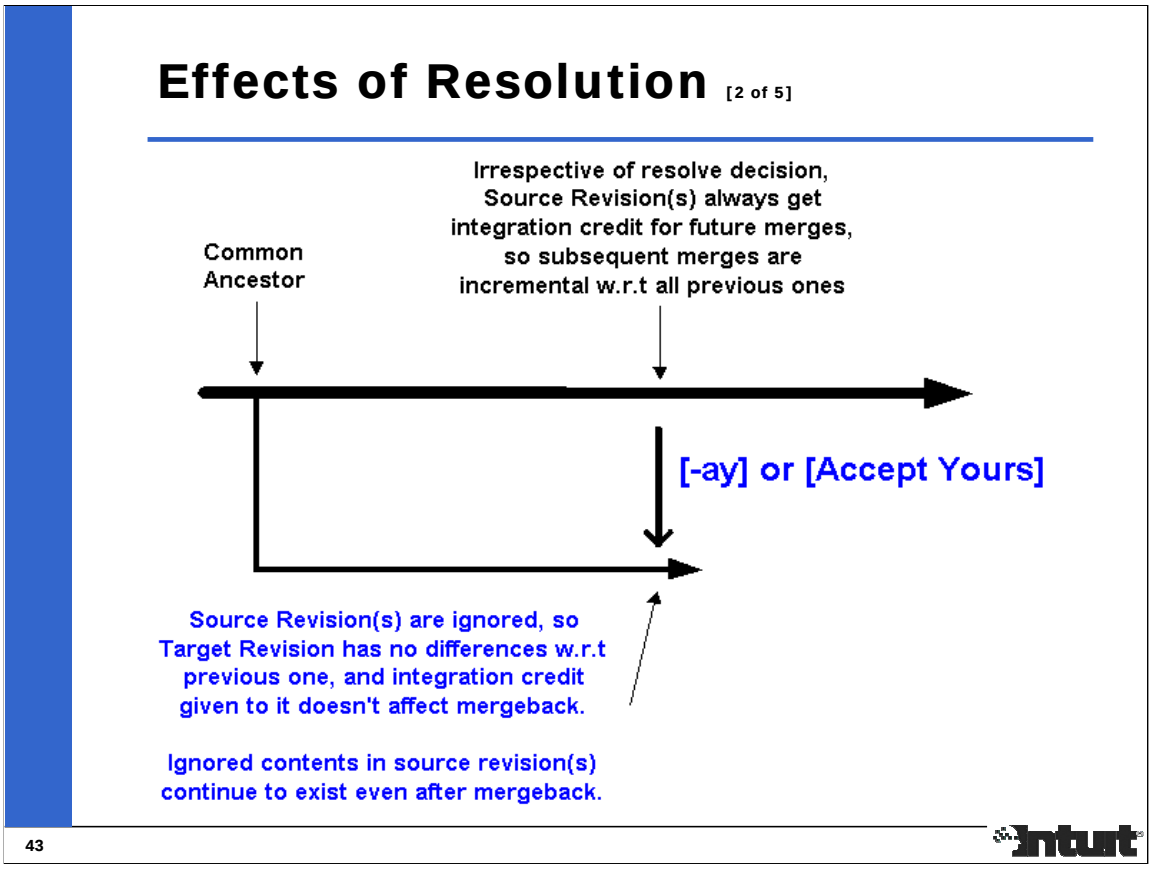
Intuit

Notes:



Notes:

[-at] or [Accept Theirs] makes the source and target versions the same,
moves the base version forward on both branches,
so gives integration credit on both branches until the source and target versions.



Notes:

[-ay] or [Accept Yours] ignores the source version(s),
moves the base version on the source branch,
so gives integration credit on the source branch until the source version
unless there are unintegrated (uncredited) revisions before the ignored version(s).

Since the target version has no diffs, it's integration credit doesn't affect mergeback;
contents of source versions continue to exist on the source branch even after a mergeback from the target branch.

This often affects a release branch which has had changes that were ignored during mergeback and is reused for the next milestone by getting integrated from it's parent development branch. "p4 diff2 [-q]" can be used to compare the contents of the 2 branches at the point where they are supposed to have the same content.

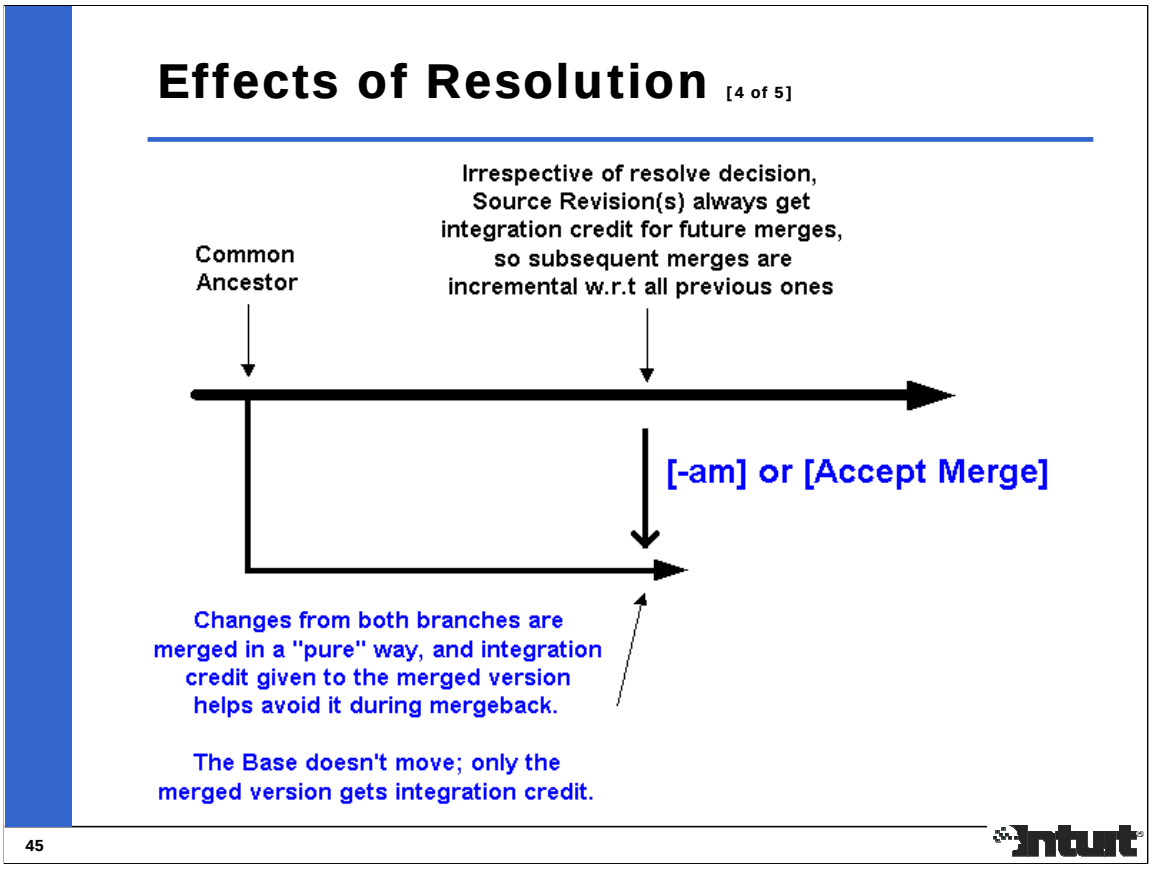
Effects of Resolution [3 of 5]

- [-as] or [Safe Automatic Resolve] behaves like [-at] or [-ay] in cases where changes existed only on source branch [-at], or changes on one branch subsume those from the other [-at | -ay]
- The effect of [-as] resolution is same as that of either [-at] or [-as]

44

intuit

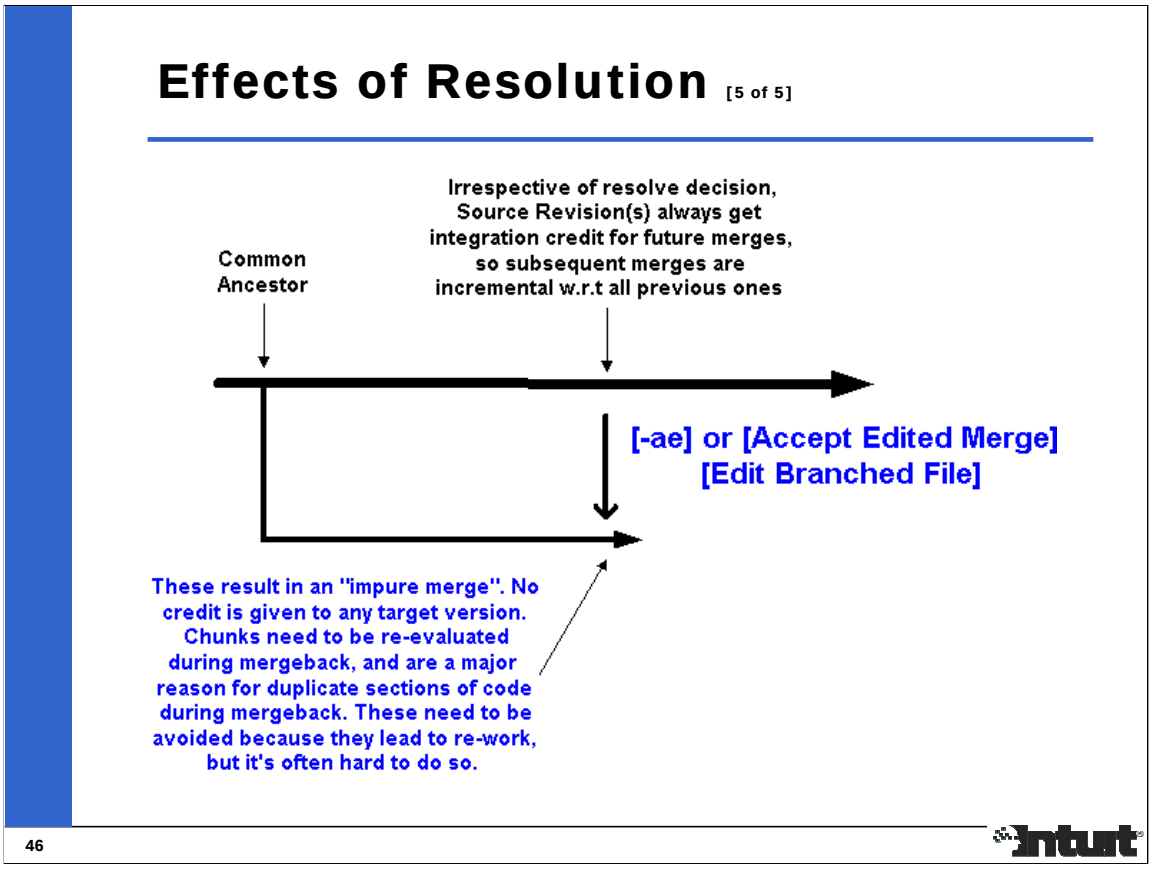
Notes:



Notes:

[-am] or [Accept Merged] merges changes from both branches in a "pure" way,
moves the base on the source branch unless there are unintegrated (uncredited) revisions before the merge, and
gives integration credit to only the merged version on the target branch.

This integration credit is identical to that of [-ay],
but doesn't cause one branch to be different from the other after mergeback.



Notes:

[-ae] or [Accept Edited Merge] and “add into” [Edit Branched File] result in an “impure merge”, moves the base on the source branch,
and gives no integration credit to the merged version on the target branch; such edits need to be re-evaluated during mergeback.

This resolve action causes conflicts during mergeback from a branch this is already known to subsume the other (be a superset of changes) because Perforce tries to be conservative about merging back resolves that were edited. Such conflicts during mergeback can be resolved as [-at] after reviewing the integration history.

There is a tradeoff between the benefits of maintaining a “pure merge”, which avoids conflicts in future mergebacks, and the cost of resolving conflicts and editing changes in separate Changelists.

“p4 resolve –af”, which forces accept on a merged file with conflicts, also results in an “impure merge”.

Integration records

- **Integration records are displayed by p4 filelog (Revision History …) and P4V's Revision Graph at source and target version**
  - branch into/from (initial branchpoint from a local depot)
  - import into/from (initial branchpoint from a remote depot)
  - copy into/from (resolve -at)
  - ignored by/ ignored (resolve -ay)
  - merge into/from (resolve -am)
  - delete into/from (integrate deleted file)
  - add into/from (open for branch/edit the file)
  - edit into/from (open for integrate/edit the file)
- **These records (actions) are at a file level, and not finer grain within a file**

47

Intuit

Notes:

Miscellaneous ideas

- **Resolve decisions are not based on when the contributor versions were submitted, but on contents of contributor versions**
- **A source version may introduce a conflict with a target version that was submitted before earlier merges into the target branch**
- **Conflict resolution in one direction doesn't mean there won't be conflicts when merging in the other direction because Perforce forces re-evaluation of "accept edit" resolve action**
- **Resolving smaller conflicts before larger ones, and resolving some files before others can help work your way through a merge because differences in resolved files can be reviewed**

48

intuit

Notes:

Merging needs to be coordinated and tracked;
small teams work better, and knowledge is lost when people drop in and out.

Mergeback from release branches lose importance or cause re-work over time because of obsolete or reimplemented code. Obsolete code can be ignored using "p4 resolve –ay". Re-work needs to be edited into the target branch.

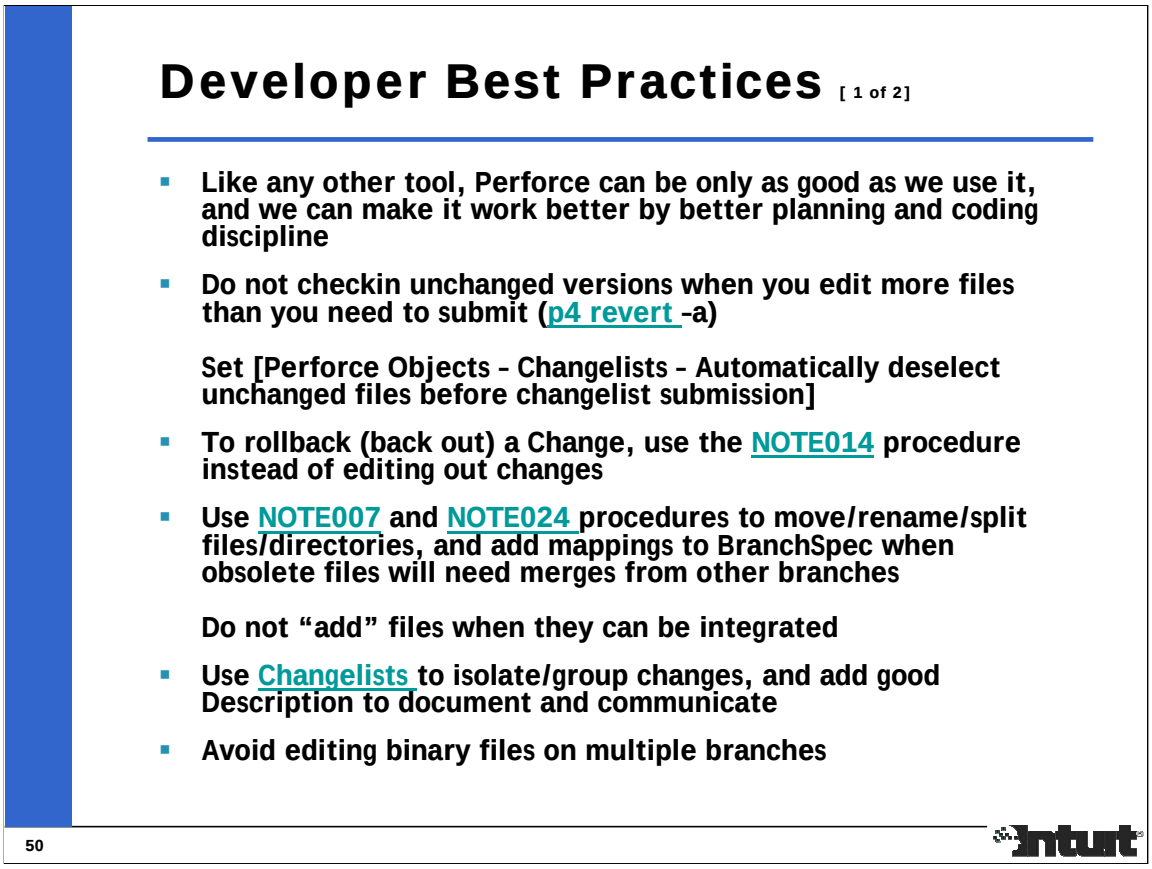
Agenda [5 of 5]

- QuickBooks Development Environment
- Procedure for merging between branches
- Inter-File Branching Algorithm
  used by Perforce to set up files for merging
- Conflict resolution, and
  its effect on future merges
- Best Practices and
  New features in Perforce v2004.2
  that aid merging
- Q&A

49

intuit

Notes:



Notes:

The P4Win setting [Perforce Objects – Changelists – Automatically deselect unchanged files before changelist submission] prevents unchanged versions from getting submitted and impacting merges. From the command line, use "p4 revert –a". For filetype changes only, use "p4 submit", or unset this option.

When conflicting changes get made on different branches, it's important to document and communicate that information in the form of comments in the code or Change Description so that it can be referred to at the time of merging between those branches.

Developer Best Practices [2 of 2]

- **Make original changes on the branch where they are needed and has evolved the least since branching.**

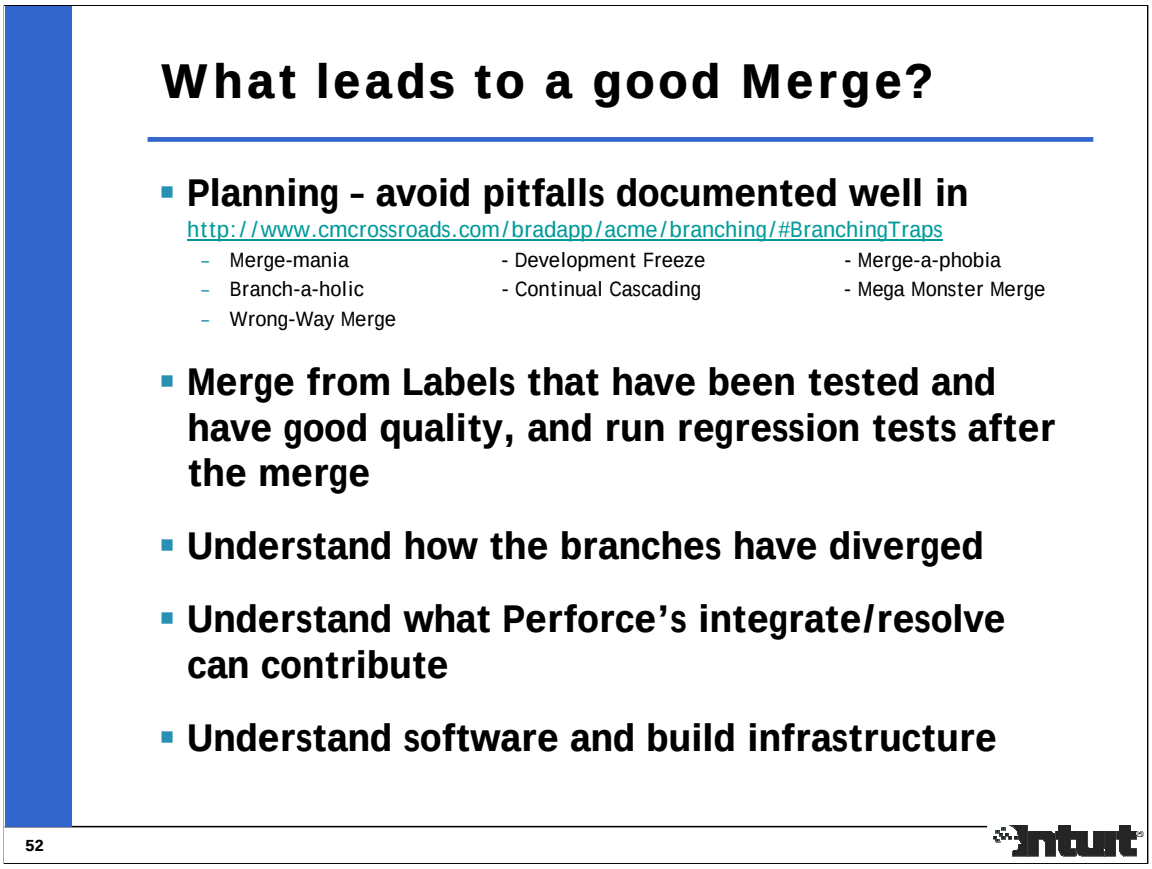
  Follow process guidelines set for target branch, and do not make exceptions without prior communication/approval
- **Don't share workspaces or usernames; doing so confuses history and process**
- **No "double/multiple checkins"; merge existing Change, and then edit, if necessary**
- **Do not copy files from one branch to another; this might add more or overwrite other changes**
- **Don't work outside of managed workspaces; cost of Branching and Merging cannot justify a Bad Practice**
- **When integrating Changes between branches, do not make build/bug fixes and unrelated white space and cosmetic changes (tabs, comments, indentation, etc) in the same Change because they cause "impure merges" and conflicts during mergeback**

51

intuit

Notes:

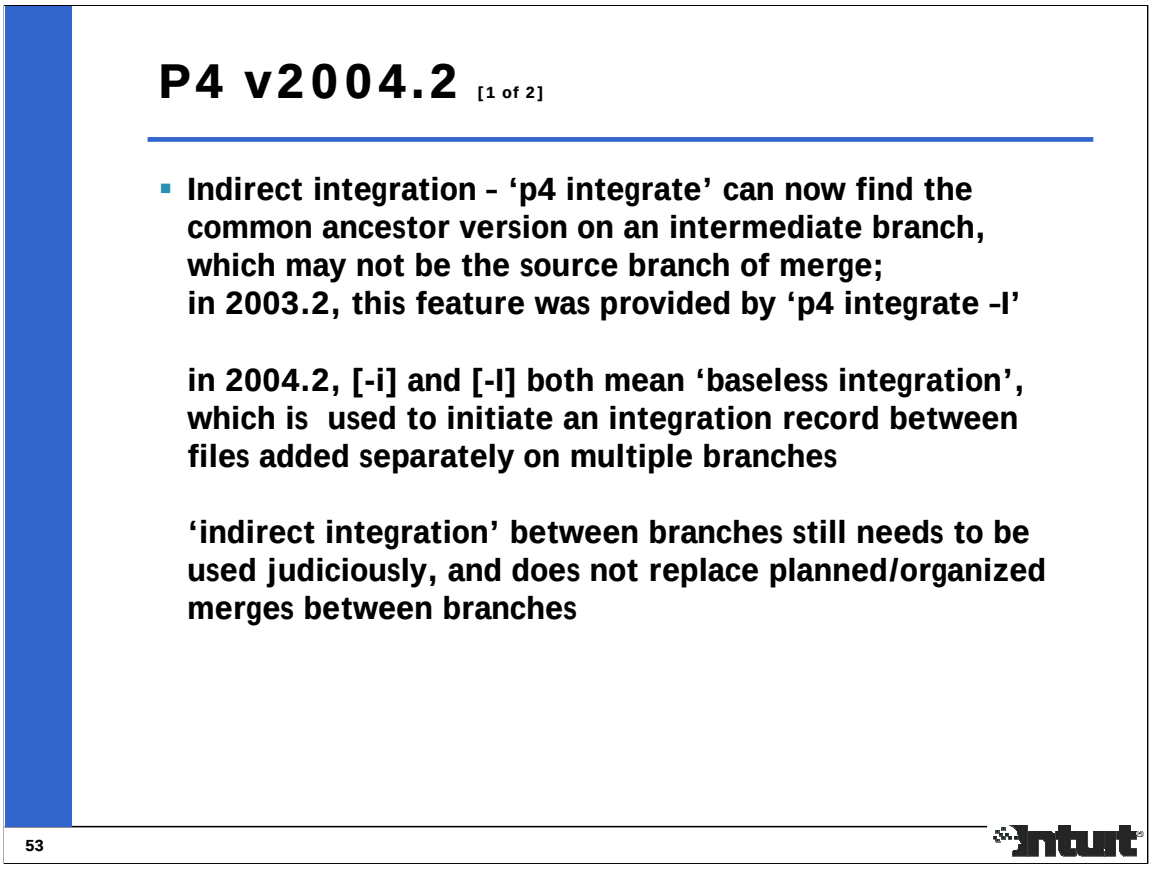
Code Champions and Architects are now responsible for making sure these guidelines are being followed.



Notes:

Longer you wait for a merge, harder it becomes to do it; complexity increases with duration of isolated development on branches.

Evaluate cost of merging before creating each branch.



Notes:

Prior to the v2003.2 release, Perforce could only merge between branches that had a parent-child relationship. The "p4 integrate –i" option enabled "baseless merges" between branches that didn't have a parent-child relationship, which could only get resolved by accepting one of two contributing versions.

In the v2003.2 release, the "p4 integrate –I" option was introduced, and it enabled merges between branches that didn't have a direct parent-child relationship. In the v2004.2 release, this is the default behavior of "p4 integrate", but this feature should be used sparingly in a branching model that should still prefer merging between branches that have a direct parent-child relationship so that there is an organized flow of changes from one branch to another.

These features are supported by the other client interfaces too.

See the reference to "driving through hedges" in "The Flow of Change" presentation at SD West 2005 Conference by Laura Wingerd, Vice President of Product Technology, Perforce Software.

P4 v2004.2 [2 of 2]

- **'p4 integrate -o' and 'p4 resolve -o' display the base version of merge**
- **'p4 integrate -Di' ignores that a source file had been deleted and readded when looking for an integration base**
- **Since the underlying support for branching is same as that for splitting/moving/renaming files, indirect integration makes it easier to merge bugfixes from older branches into a branch having a reorganized source structure**

  To enable indirect integration to merge bugfixes from older branches into a branch having a reorganized source structure, it is critical that 'p4 rename' be used to move/rename files, and 'p4 integrate' be used to split files
- **Improved resolve logic reduces conflicting regions with more complex detection of commonality of changes**

54

intuit

Notes:

Note that "p4 rename" is actually 3 commands – integrate, delete, and submit.

Along with 'Revision Graph', these features bring Perforce's support for parallel development closer to that of ClearCase.

P4Win and P4V v2004.2

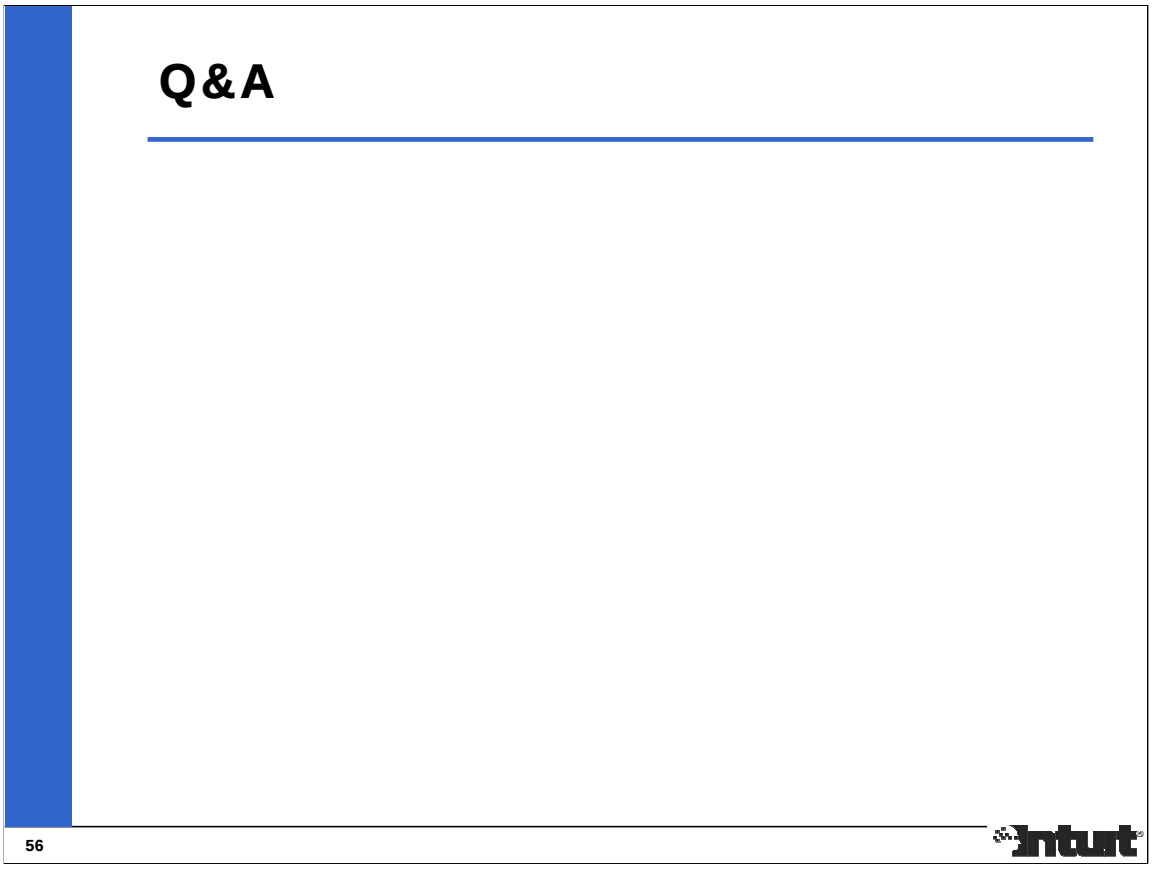
- **Revision Graph displays revision history visually, similar to ClearCase's "lsvtree -graphical" and merge hyperlinks**
- **Time-lapse View displays evolution of file's contents over time**
- **"Revision Graph…" and "Time-lapse View…" menu items are enabled in P4Win if P4V is also installed**
- **In P4Win, [Right click on filename - View (depot version) using - Annotations using Rev#s…] to annotate contents of a file version**
- **P4V's File revision history includes integration and label history**
- **P4V has Built-in differencing and 3-way merge tool**

55

intuit

Notes:

See: http://www.perforce.com/perforce/products/p4win.html
See: http://www.perforce.com/perforce/products/p4v.html
See: http://www.perforce.com/perforce/doc.042/manuals/p4win-gs/p4win-gs.pdf



Notes:

Thanks to the following people for reviewing this presentation and their feedback:

Bruce Wobbe, Intuit
Sam Stafford, Perforce Software
Steve Smith, Intuit
Vinay Shitikond, Microsoft