QPDF Manual

For QPDF Version 4.1.0, April 14, 2013

Jay Berkenbilt

QPDF Manual: For QPDF Version 4.1.0, April 14, 2013

Jay Berkenbilt

Table of Contents

General Information

QPDF is a program that does structural, content-preserving transformations on PDF files. QPDF's website is located at http://qpdf.sourceforge.net/. QPDF's source code is hosted on github at https://github.com/qpdf/qpdf.

QPDF has been released under the terms of Version 2.0 of the Artistic License [http://www.opensource.org/licenses/artistic-license-2.0.php], a copy of which appears in the file *Artistic-2.0* in the source distribution.

QPDF was originally created in 2001 and modified periodically between 2001 and 2005 during my employment at Apex CoVantage [http://www.apexcovantage.com]. Upon my departure from Apex, the company graciously allowed me to take ownership of the software and continue maintaining as an open source project, a decision for which I am very grateful. I have made considerable enhancements to it since that time. I feel fortunate to have worked for people who would make such a decision. This work would not have been possible without their support.

Chapter 1. What is QPDF?

QPDF is a program that does structural, content-preserving transformations on PDF files. It could have been called something like *pdf-to-pdf*. It also provides many useful capabilities to developers of PDF-producing software or for people who just want to look at the innards of a PDF file to learn more about how they work.

With QPDF, it is possible to copy objects from one PDF file into another and to manipulate the list of pages in a PDF file. This makes it possible to merge and split PDF files. The QPDF library also makes it possible for you to create PDF files from scratch. In this mode, you are responsible for supplying all the contents of the file, while the QPDF library takes care off all the syntactical representation of the objects, creation of cross references tables and, if you use them, object streams, encryption, linearization, and other syntactic details. You are still responsible for generating PDF content on your own.

QPDF has been designed with very few external dependencies, and it is intentionally very lightweight. QPDF is *not* a PDF content creation library, a PDF viewer, or a program capable of converting PDF into other formats. In particular, QPDF knows nothing about the semantics of PDF content streams. If you are looking for something that can do that, you should look elsewhere. However, once you have a valid PDF file, QPDF can be used to transform that file in ways perhaps your original PDF creation can't handle. For example, many programs generate simple PDF files but can't password-protect them, web-optimize them, or perform other transformations of that type.

Chapter 2. Building and Installing QPDF

This chapter describes how to build and install qpdf. Please see also the *README* and *INSTALL* files in the source distribution.

2.1. System Requirements

The qpdf package has relatively few external dependencies. In order to build qpdf, the following packages are required:

- zlib: http://www.zlib.net/

- pcre: http://www.pcre.org/

- gnu make 3.81 or newer: http://www.gnu.org/software/make

- perl version 5.8 or newer: http://www.perl.org/; required for **fix-qdf** and the test suite.

- GNU diffutils (any version): http://www.gnu.org/software/diffutils/ is required to run the test suite. Note that this is the version of diff present on virtually all GNU/Linux systems. This is required because the test suite uses **diff -u**.

- A C++ compiler that works well with STL and has the `long long` type. Most modern C++ compilers should fit the bill fine. QPDF is tested with gcc and Microsoft Visual C++.

Part of qpdf's test suite does comparisons of the contents PDF files by converting them images and comparing the images. The image comparison tests are disabled by default. Those tests are not required for determining correctness of a qpdf build if you have not modified the code since the test suite also contains expected output files that are compared literally. The image comparison tests provide an extra check to make sure that any content transformations don't break the rendering of pages. Transformations that affect the content streams themselves are off by default and are only provided to help developers look into the contents of PDF files. If you are making deep changes to the library that cause changes in the contents of the files that qpdf generates, then you should enable the image comparison tests. Enable them by running **configure** with the **--enable-test-compare-images** flag. If you enable this, the following additional requirements are required by the test suite. Note that in no case are these items required to use qpdf.

- libtiff: http://www.remotesensing.org/libtiff/

- GhostScript version 8.60 or newer: http://www.ghostscript.com

If you do not enable this, then you do not need to have tiff and ghostscript.

If Adobe Reader is installed as **acroread**, some additional test cases will be enabled. These test cases simply verify that Adobe Reader can open the files that qpdf creates. They require version 8.0 or newer to pass. However, in order to avoid having qpdf depend on non-free (as in liberty) software, the test suite will still pass without Adobe reader, and the test suite still exercises the full functionality of the software.

Pre-built documentation is distributed with qpdf, so you should generally not need to rebuild the documentation. In order to build the documentation from its docbook sources, you need the docbook XML style sheets (http://downloads.sourceforge.net/docbook/). To build the PDF version of the documentation, you need Apache fop (http://xml.apache.org/fop/) version 0.94 or higher.

2.2. Build Instructions

Building qpdf on UNIX is generally just a matter of running

```
./configure
make
```

You can also run **make check** to run the test suite and **make install** to install. Please run **./configure --help** for options on what can be configured. You can also set the value of *DESTDIR* during installation to install to a temporary location, as is common with many open source packages. Please see also the *README* and *INSTALL* files in the source distribution.

Building on Windows is a little bit more complicated. For details, please see *README-windows.txt* in the source distribution. You can also download a binary distribution for Windows. There is a port of qpdf to Visual C++ version 6 in the *contrib* area generously contributed by Jian Ma. This is also discussed in more detail in *README-windows.txt*.

There are some other things you can do with the build. Although qpdf uses autoconf, it does not use automake but instead uses a hand-crafted non-recursive Makefile that requires gnu make. If you're really interested, please read the comments in the top-level *Makefile*.

Chapter 3. Running QPDF

This chapter describes how to run the qpdf program from the command line.

3.1. Basic Invocation

When running qpdf, the basic invocation is as follows:

```
qpdf [ options ] infilename [ outfilename ]
```

This converts PDF file **infilename** to PDF file **outfilename**. The output file is functionally identical to the input file but may have been structurally reorganized. Also, orphaned objects will be removed from the file. Many transformations are available as controlled by the options below. In place of **infilename**, the parameter **--empty** may be specified. This causes qpdf to use a dummy input file that contains zero pages. The only normal use case for using **--empty** would be if you were going to add pages from another source, as discussed in Section 3.4, "Page Selection Options", page 6.

outfilename does not have to be seekable, even when generating linearized files. Specifying "**-**" as **outfilename** means to write to standard output. However, you can't specify the same file as both the input and the output because qpdf reads data from the input file as it writes to the output file.

Most options require an output file, but some testing or inspection commands do not. These are specifically noted.

3.2. Basic Options

The following options are the most common ones and perform commonly needed transformations.

--password=password
: Specifies a password for accessing encrypted files.

--linearize
: Causes generation of a linearized (web-optimized) output file.

--copy-encryption=file
: Encrypt the file using the same encryption parameters, including user and owner password, as the specified file. Use **--encrypt-file-password** to specify a password if one is needed to open this file. Note that copying the encryption parameters from a file also copies the first half of /ID from the file since this is part of the encryption parameters.

--encrypt-file-password=password
: If the file specified with **--copy-encryption** requires a password, specify the password using this option. Note that only one of the user or owner password is required. Both passwords will be preserved since QPDF does not distinguish between the two passwords. It is possible to preserve encryption parameters, including the owner password, from a file even if you don't know the file's owner password.

--encrypt options --
: Causes generation an encrypted output file. Please see Section 3.3, "Encryption Options," page 5 for details on how to specify encryption parameters.

--decrypt
: Removes any encryption on the file. A password must be supplied if the file is password protected.

--pages options --
: Select specific pages from one or more input files. See Section 3.4, "Page Selection Options," page 6 for details on how to do page selection (splitting and merging).

Password-protected files may be opened by specifying a password. By default, qpdf will preserve any encryption data associated with a file. If **--decrypt** is specified, qpdf will attempt to remove any encryption information. If **--encrypt** is specified, qpdf will replace the document's encryption parameters with whatever is specified.

Note that qpdf does not obey encryption restrictions already imposed on the file. Doing so would be meaningless since qpdf can be used to remove encryption from the file entirely. This functionality is not intended to be used for bypassing copyright restrictions or other restrictions placed on files by their producers.

In all cases where qpdf allows specification of a password, care must be taken if the password contains characters that fall outside of the 7-bit US-ASCII character range to ensure that the exact correct byte sequence is provided. It is possible that a future version of qpdf may handle this more gracefully. For example, if a password was encrypted using a password that was encoded in ISO-8859-1 and your terminal is configured to use UTF-8, the password you supply may not work properly. There are various approaches to handling this. For example, if you are using Linux and have the iconv executable (part of the ICU package) installed, you could pass **--password=`echo *password* | iconv -t iso-8859-1`** to qpdf where *password* is a password specified in your terminal's locale. A detailed discussion of this is out of scope for this manual, but just be aware of this issue if you have trouble with a password that contains 8-bit characters.

3.3. Encryption Options

To change the encryption parameters of a file, use the --encrypt flag. The syntax is

```
--encrypt user-password owner-password key-length [ restrictions ] --
```

Note that "**--**" terminates parsing of encryption flags and must be present even if no restrictions are present.

Either or both of the user password and the owner password may be empty strings.

The value for **key-length** may be 40, 128, or 256. The restriction flags are dependent upon key length. When no additional restrictions are given, the default is to be fully permissive.

If **key-length** is 40, the following restriction options are available:

--print=[yn]
 Determines whether or not to allow printing.

--modify=[yn]
 Determines whether or not to allow document modification.

--extract=[yn]
 Determines whether or not to allow text/image extraction.

--annotate=[yn]
 Determines whether or not to allow comments and form fill-in and signing.

If **key-length** is 128, the following restriction options are available:

--accessibility=[yn]
 Determines whether or not to allow accessibility to visually impaired.

--extract=[yn]
 Determines whether or not to allow text/graphic extraction.

--print=*print-opt*
 Controls printing access. *print-opt* may be one of the following:

- **full**: allow full printing

- **low**: allow low-resolution printing only

- **none**: disallow printing

--modify=`modify-opt`
Controls modify access. **`modify-opt`** may be one of the following, each of which implies all the options that follow it:

- **all**: allow full document modification

- **annotate**: allow comment authoring and form operations

- **form**: allow form field fill-in and signing

- **assembly**: allow document assembly only

- **none**: allow no modifications

--cleartext-metadata
If specified, any metadata stream in the document will be left unencrypted even if the rest of the document is encrypted. This also forces the PDF version to be at least 1.5.

--use-aes=[yn]
If **--use-aes=y** is specified, AES encryption will be used instead of RC4 encryption. This forces the PDF version to be at least 1.6.

--force-V4
Use of this option forces the /V and /R parameters in the document's encryption dictionary to be set to the value 4. As qpdf will automatically do this when required, there is no reason to ever use this option. It exists primarily for use in testing qpdf itself. This option also forces the PDF version to be at least 1.5.

If **`key-length`** is 256, the minimum PDF version is 1.7 with extension level 8, and the AES-based encryption format used is the PDF 2.0 encryption method supported by Acrobat X. the same options are available as with 128 bits with the following exceptions:

--use-aes
This option is not available with 256-bit keys. AES is always used with 256-bit encryption keys.

--force-V4
This option is not available with 256 keys.

--force-R5
If specified, qpdf sets the minimum version to 1.7 at extension level 3 and writes the deprecated encryption format used by Acrobat version IX. This option should not be used in practice to generate PDF files that will be in general use, but it can be useful to generate files if you are trying to test proper support in another application for PDF files encrypted in this way.

The default for each permission option is to be fully permissive.

3.4. Page Selection Options

Starting with qpdf 3.0, it is possible to split and merge PDF files by selecting pages from one or more input files. Whatever file is given as the primary input file is used as the starting point, but its pages are replaced with pages as specified.

```
--pages input-file [ --password=password ] page-range [ ... ] --
```

Multiple input files may be specified. Each one is given as the name of the input file, an optional password (if required to open the file), and the range of pages. Note that "**--**" terminates parsing of page selection flags.

For each file that pages should be taken from, specify the file, a password needed to open the file (if any), and a page range. The password needs to be given only once per file. If any of the input files are the same as the primary input file or the file used to copy encryption parameters (if specified), you do not need to repeat the password here. The same file can be repeated multiple times. If a file that is repeated has a password, the password only has to be given the first time. All non-page data (info, outlines, page numbers, etc.) are taken from the primary input file. To discard these, use **--empty** as the primary input.

It is not presently possible to specify the same page from the same file directly more than once, but you can make this work by specifying two different paths to the same file (such as by putting ./ somewhere in the path). This can also be used if you want to repeat a page from one of the input files in the output file. This may be made more convenient in a future version of qpdf if there is enough demand for this feature.

The page range is a set of numbers separated by commas, ranges of numbers separated dashes, or combinations of those. The character "z" represents the last page. Pages can appear in any order. Ranges can appear with a high number followed by a low number, which causes the pages to appear in reverse. Repeating a number will cause an error, but you can use the workaround discussed above should you really want to include the same page twice.

Example page ranges:

- `1,3,5-9,15-12`: pages 1, 2, 3, 5, 6, 7, 8, 9, 15, 14, 13, and 12.

- `z-1`: all pages in the document in reverse

Note that qpdf doesn't presently do anything special about other constructs in a PDF file that may know about pages, so semantics of splitting and merging vary across features. For example, the document's outlines (bookmarks) point to actual page objects, so if you select some pages and not others, bookmarks that point to pages that are in the output file will work, and remaining bookmarks will not work. On the other hand, page labels (page numbers specified in the file) are just sequential, so page labels will be messed up in the output file. A future version of **qpdf** may do a better job at handling these issues. (Note that the qpdf library already contains all of the APIs required in order to implement this in your own application if you need it.) In the mean time, you can always use **--empty** as the primary input file to avoid copying all of that from the first file. For example, to take pages 1 through 5 from a *infile.pdf* while preserving all metadata associated with that file, you could use

```
qpdf infile.pdf --pages infile.pdf 1-5 -- outfile.pdf
```

If you wanted pages 1 through 5 from *infile.pdf* but you wanted the rest of the metadata to be dropped, you could instead run

```
qpdf --empty --pages infile.pdf 1-5 -- outfile.pdf
```

If you wanted to take pages 1–5 from *file1.pdf* and pages 11–15 from *file2.pdf* in reverse, you would run

```
qpdf file1.pdf --pages file1.pdf 1-5 file2.pdf 15-11 -- outfile.pdf
```

If, for some reason, you wanted to take the first page of an encrypted file called *encrypted.pdf* with password `pass` and repeat it twice in an output file, and if you wanted to drop metadata (like page numbers and outlines) but preserve encryption, you would use

```
qpdf --empty --copy-encryption=encrypted.pdf --encryption-file-password=pass
--pages encrypted.pdf --password=pass 1 ./encrypted.pdf --password=pass 1 --
```

```
outfile.pdf
```

Note that we had to specify the password all three times because giving a password as **--encryption-file-password** doesn't count for page selection, and as far as qpdf is concerned, *encrypted.pdf* and *./encrypted.pdf* are separated files. These are all corner cases that most users should hopefully never have to be bothered with.

3.5. Advanced Transformation Options

These transformation options control fine points of how qpdf creates the output file. Mostly these are of use only to people who are very familiar with the PDF file format or who are PDF developers. The following options are available:

--stream-data=*option*
    Controls transformation of stream data. The value of *option* may be one of the following:

  - **compress**: recompress stream data when possible (default)

  - **preserve**: leave all stream data as is

  - **uncompress**: uncompress stream data when possible

--normalize-content=[yn]
    Enables or disables normalization of content streams.

--suppress-recovery
    Prevents qpdf from attempting to recover damaged files.

--object-streams=*mode*
    Controls handing of object streams. The value of *mode* may be one of the following:

  - **preserve**: preserve original object streams (default)

  - **disable**: don't write any object streams

  - **generate**: use object streams wherever possible

--ignore-xref-streams
    Tells qpdf to ignore any cross-reference streams.

--qdf
    Turns on QDF mode. For additional information on QDF, please see Chapter 4, *QDF Mode*, page 12.

--min-version=*version*
    Forces the PDF version of the output file to be at least *version*. In other words, if the input file has a lower version than the specified version, the specified version will be used. If the input file has a higher version, the input file's original version will be used. It is seldom necessary to use this option since qpdf will automatically increase the version as needed when adding features that require newer PDF readers.

    The version number may be expressed in the form *major.minor.extension-level*, in which case the version is interpreted as *major.minor* at extension level *extension-level*. For example, version 1.7.8 represents version 1.7 at extension level 8. Note that minimal syntax checking is done on the command line.

--force-version=*version*
    This option forces the PDF version to be the exact version specified *even when the file may have content that is not supported in that version*. The version number is interpreted in the same way as with **--min-version** so that extension levels can be set. In some cases, forcing the output file's PDF version to be lower than that of the input file will cause qpdf to disable certain features of the document. Specifically, 256-bit keys are disabled if

the version is less than 1.7 with extension level 8 (except R5 is disabled if less than 1.7 with extension level 3), AES encryption is disabled if the version is less than 1.6, cleartext metadata and object streams are disabled if less than 1.5, 128-bit encryption keys are disabled if less than 1.4, and all encryption is disabled if less than 1.3. Even with these precautions, qpdf won't be able to do things like eliminate use of newer image compression schemes, transparency groups, or other features that may have been added in more recent versions of PDF.

As a general rule, with the exception of big structural things like the use of object streams or AES encryption, PDF viewers are supposed to ignore features in files that they don't support from newer versions. This means that forcing the version to a lower version may make it possible to open your PDF file with an older version, though bear in mind that some of the original document's functionality may be lost.

By default, when a stream is encoded using non-lossy filters that qpdf understands and is not already compressed using a good compression scheme, qpdf will uncompress and recompress streams. Assuming proper filter implements, this is safe and generally results in smaller files. This behavior may also be explicitly requested with **--stream-data=compress**.

When **--stream-data=preserve** is specified, qpdf will never attempt to change the filtering of any stream data.

When **--stream-data=uncompress** is specified, qpdf will attempt to remove any non-lossy filters that it supports. This includes /FlateDecode, /LZWDecode, /ASCII85Decode, and /ASCIIHexDecode. This can be very useful for inspecting the contents of various streams.

When **--normalize-content=y** is specified, qpdf will attempt to normalize whitespace and newlines in page content streams. This is generally safe but could, in some cases, cause damage to the content streams. This option is intended for people who wish to study PDF content streams or to debug PDF content. You should not use this for "production" PDF files.

Ordinarily, qpdf will attempt to recover from certain types of errors in PDF files. These include errors in the cross-reference table, certain types of object numbering errors, and certain types of stream length errors. Sometimes, qpdf may think it has recovered but may not have actually recovered, so care should be taken when using this option as some data loss is possible. The **--suppress-recovery** option will prevent qpdf from attempting recovery. In this case, it will fail on the first error that it encounters.

Object streams, also known as compressed objects, were introduced into the PDF specification at version 1.5, corresponding to Acrobat 6. Some older PDF viewers may not support files with object streams. qpdf can be used to transform files with object streams to files without object streams or vice versa. As mentioned above, there are three object stream modes: **preserve**, **disable**, and **generate**.

In **preserve** mode, the relationship to objects and the streams that contain them is preserved from the original file. In **disable** mode, all objects are written as regular, uncompressed objects. The resulting file should be readable by older PDF viewers. (Of course, the content of the files may include features not supported by older viewers, but at least the structure will be supported.) In **generate** mode, qpdf will create its own object streams. This will usually result in more compact PDF files, though they may not be readable by older viewers. In this mode, qpdf will also make sure the PDF version number in the header is at least 1.5.

Ordinarily, qpdf reads cross-reference streams when they are present in a PDF file. If **--ignore-xref-streams** is specified, qpdf will ignore any cross-reference streams for hybrid PDF files. The purpose of hybrid files is to make some content available to viewers that are not aware of cross-reference streams. It is almost never desirable to ignore them. The only time when you might want to use this feature is if you are testing creation of hybrid PDF files and wish to see how a PDF consumer that doesn't understand object and cross-reference streams would interpret such a file.

The **--qdf** flag turns on QDF mode, which changes some of the defaults described above. Specifically, in QDF mode, by default, stream data is uncompressed, content streams are normalized, and encryption is removed. These defaults can still be overridden by specifying the appropriate options as described above. Additionally, in QDF mode, stream lengths are stored as indirect objects, objects are laid out in a less efficient but more readable fashion, and the documents are interspersed with comments that make it easier for the user to find things and also make it possible for **fix-qdf** to

work properly. QDF mode is intended for people, mostly developers, who wish to inspect or modify PDF files in a text editor. For details, please see Chapter 4, *QDF Mode*, page 12.

3.6. Testing, Inspection, and Debugging Options

These options can be useful for digging into PDF files or for use in automated test suites for software that uses the qpdf library. When any of the options in this section are specified, no output file should be given. The following options are available:

--static-id

Causes generation of a fixed value for /ID. This is intended for testing only. Never use it for production files.

--static-aes-iv

Causes use of a static initialization vector for AES-CBC. This is intended for testing only so that output files can be reproducible. Never use it for production files. This option in particular is not secure since it significantly weakens the encryption.

--no-original-object-ids

Suppresses inclusion of original object ID comments in QDF files. This can be useful when generating QDF files for test purposes, particularly when comparing them to determine whether two PDF files have identical content.

-show-encryption

Shows document encryption parameters. Also shows the document's user password if the owner password is given.

-check-linearization

Checks file integrity and linearization status.

-show-linearization

Checks and displays all data in the linearization hint tables.

-show-xref

Shows the contents of the cross-reference table in a human-readable form. This is especially useful for files with cross-reference streams which are stored in a binary format.

-show-object=obj[,gen]

Show the contents of the given object. This is especially useful for inspecting objects that are inside of object streams (also known as "compressed objects").

-raw-stream-data

When used along with the **--show-object** option, if the object is a stream, shows the raw stream data instead of object's contents.

-filtered-stream-data

When used along with the **--show-object** option, if the object is a stream, shows the filtered stream data instead of object's contents. If the stream is filtered using filters that qpdf does not support, an error will be issued.

-show-pages

Shows the object and generation number for each page dictionary object and for each content stream associated with the page. Having this information makes it more convenient to inspect objects from a particular page.

-with-images

When used along with **--show-pages**, also shows the object and generation numbers for the image objects on each page. (At present, information about images in shared resource dictionaries are not output by this command. This is discussed in a comment in the source code.)

-check

    Checks file structure and well as encryption, linearization, and encoding of stream data. A file for which **--check** reports no errors may still have errors in stream data content but should otherwise be structurally sound. If **--check** any errors, qpdf will exit with a status of 2. There are some recoverable conditions that **--check** detects. These are issued as warnings instead of errors. If qpdf finds no errors but finds warnings, it will exit with a status of 3 (as of version 2.0.4).

The **--raw-stream-data** and **--filtered-stream-data** options are ignored unless **--show-object** is given. Either of these options will cause the stream data to be written to standard output. In order to avoid commingling of stream data with other output, it is recommend that these objects not be combined with other test/inspection options.

If **--filtered-stream-data** is given and **--normalize-content=y** is also given, qpdf will attempt to normalize the stream data as if it is a page content stream. This attempt will be made even if it is not a page content stream, in which case it will produce unusable results.

Chapter 4. QDF Mode

In QDF mode, qpdf creates PDF files in what we call *QDF form*. A PDF file in QDF form, sometimes called a QDF file, is a completely valid PDF file that has `%QDF-1.0` as its third line (after the pdf header and binary characters) and has certain other characteristics. The purpose of QDF form is to make it possible to edit PDF files, with some restrictions, in an ordinary text editor. This can be very useful for experimenting with different PDF constructs or for making one-off edits to PDF files (though there are other reasons why this may not always work).

It is ordinarily very difficult to edit PDF files in a text editor for two reasons: most meaningful data in PDF files is compressed, and PDF files are full of offset and length information that makes it hard to add or remove data. A QDF file is organized in a manner such that, if edits are kept within certain constraints, the **fix-qdf** program, distributed with qpdf, is able to restore edited files to a correct state. The **fix-qdf** program takes no command-line arguments. It reads a possibly edited QDF file from standard input and writes a repaired file to standard output.

The following attributes characterize a QDF file:

- All objects appear in numerical order in the PDF file, including when objects appear in object streams.

- Objects are printed in an easy-to-read format, and all line endings are normalized to UNIX line endings.

- Unless specifically overridden, streams appear uncompressed (when qpdf supports the filters and they are compressed with a non-lossy compression scheme), and most content streams are normalized (line endings are converted to just a UNIX-style linefeeds).

- All streams lengths are represented as indirect objects, and the stream length object is always the next object after the stream. If the stream data does not end with a newline, an extra newline is inserted, and a special comment appears after the stream indicating that this has been done.

- If the PDF file contains object streams, if object stream *n* contains *k* objects, those objects are numbered from *n +1* through *n+k*, and the object number/offset pairs appear on a separate line for each object. Additionally, each object in the object stream is preceded by a comment indicating its object number and index. This makes it very easy to find objects in object streams.

- All beginnings of objects, `stream` tokens, `endstream` tokens, and `endobj` tokens appear on lines by themselves. A blank line follows every `endobj` token.

- If there is a cross-reference stream, it is unfiltered.

- Page dictionaries and page content streams are marked with special comments that make them easy to find.

- Comments precede each object indicating the object number of the corresponding object in the original file.

When editing a QDF file, any edits can be made as long as the above constraints are maintained. This means that you can freely edit a page's content without worrying about messing up the QDF file. It is also possible to add new objects so long as those objects are added after the last object in the file or subsequent objects are renumbered. If a QDF file has object streams in it, you can always add the new objects before the xref stream and then change the number of the xref stream, since nothing generally ever references it by number.

It is not generally practical to remove objects from QDF files without messing up object numbering, but if you remove all references to an object, you can run qpdf on the file (after running **fix-qdf**), and qpdf will omit the now-orphaned object.

When **fix-qdf** is run, it goes through the file and recomputes the following parts of the file:

- the `/N`, `/W`, and `/First` keys of all object stream dictionaries

- the pairs of numbers representing object numbers and offsets of objects in object streams

- all stream lengths

- the cross-reference table or cross-reference stream

- the offset to the cross-reference table or cross-reference stream following the `startxref` token

Chapter 5. Using the QPDF Library

The source tree for the qpdf package has an *examples* directory that contains a few example programs. The *qpdf/ qpdf.cc* source file also serves as a useful example since it exercises almost all of the qpdf library's public interface. The best source of documentation on the library itself is reading comments in *include/qpdf/QPDF.hh*, *include/qpdf/ QDFWriter.hh*, and *include/qpdf/QPDFObjectHandle.hh*.

All header files are installed in the *include/qpdf* directory. It is recommend that you use `#include <qpdf/ QPDF.hh>` rather than adding *include/qpdf* to your include path.

When linking against the qpdf static library, you may also need to specify `-lpcre -lz` on your link command. If your system understands how to read libtool *.la* files, this may not be necessary.

The qpdf library is safe to use in a multithreaded program, but no individual `QPDF` object instance (including `QPDF`, `QPDFObjectHandle`, or `QPDFWriter`) can be used in more than one thread at a time. Multiple threads may simultaneously work with different instances of these and all other QPDF objects.

Chapter 6. Design and Library Notes

6.1. Introduction

This section was written prior to the implementation of the qpdf package and was subsequently modified to reflect the implementation. In some cases, for purposes of explanation, it may differ slightly from the actual implementation. As always, the source code and test suite are authoritative. Even if there are some errors, this document should serve as a road map to understanding how this code works.

In general, one should adhere strictly to a specification when writing but be liberal in reading. This way, the product of our software will be accepted by the widest range of other programs, and we will accept the widest range of input files. This library attempts to conform to that philosophy whenever possible but also aims to provide strict checking for people who want to validate PDF files. If you don't want to see warnings and are trying to write something that is tolerant, you can call `setSuppressWarnings(true)`. If you want to fail on the first error, you can call `setAttemptRecovery(false)`. The default behavior is to generating warnings for recoverable problems. Note that recovery will not always produce the desired results even if it is able to get through the file. Unlike most other PDF files that produce generic warnings such as "This file is damaged,", qpdf generally issues a detailed error message that would be most useful to a PDF developer. This is by design as there seems to be a shortage of PDF validation tools out there. (This was, in fact, one of the major motivations behind the initial creation of qpdf.)

6.2. Design Goals

The QPDF package includes support for reading and rewriting PDF files. It aims to hide from the user details involving object locations, modified (appended) PDF files, the directness/indirectness of objects, and stream filters including encryption. It does not aim to hide knowledge of the object hierarchy or content stream contents. Put another way, a user of the qpdf library is expected to have knowledge about how PDF files work, but is not expected to have to keep track of bookkeeping details such as file positions.

A user of the library never has to care whether an object is direct or indirect. All access to objects deals with this transparently. All memory management details are also handled by the library.

The **PointerHolder** object is used internally by the library to deal with memory management. This is basically a smart pointer object very similar in spirit to the Boost library's **shared_ptr** object, but predating it by several years. This library also makes use of a technique for giving fine-grained access to methods in one class to other classes by using public subclasses with friends and only private members that in turn call private methods of the containing class. See **QPDFObjectHandle::Factory** as an example.

The top-level qpdf class is **QPDF**. A **QPDF** object represents a PDF file. The library provides methods for both accessing and mutating PDF files.

QPDFObject is the basic PDF Object class. It is an abstract base class from which are derived classes for each type of PDF object. Clients do not interact with Objects directly but instead interact with **QPDFObjectHandle**.

QPDFObjectHandle contains **PointerHolder<QPDFObject>** and includes accessor methods that are type-safe proxies to the methods of the derived object classes as well as methods for querying object types. They can be passed around by value, copied, stored in containers, etc. with very low overhead. Instances of **QPDFObjectHandle** always contain a reference back to the **QPDF** object from which they were created. A **QPDFObjectHandle** may be direct or indirect. If indirect, the **QPDFObject** the **PointerHolder** initially points to is a null pointer. In this case, the first attempt to access the underlying **QPDFObject** will result in the **QPDFObject** being resolved via a call to the referenced **QPDF** instance. This makes it essentially impossible to make coding errors in which certain things will work for some PDF files and not for others based on which objects are direct and which objects are indirect.

Instances of **QPDFObjectHandle** can be directly created and modified using static factory methods in the **QPDFObjectHandle** class. There are factory methods for each type of object as well as a convenience method *QPDFObjectHandle::parse* that creates an object from a string representation of the object. Existing instances of **QPDFObjectHandle** can also be modified in several ways. See comments in *QPDFObjectHandle.hh* for details.

When the **QPDF** class creates a new object, it dynamically allocates the appropriate type of **QPDFObject** and immediately hands the pointer to an instance of **QPDFObjectHandle**. The parser reads a token from the current file position. If the token is a not either a dictionary or array opener, an object is immediately constructed from the single token and the parser returns. Otherwise, the parser is invoked recursively in a special mode in which it accumulates objects until it finds a balancing closer. During this process, the "R" keyword is recognized and an indirect **QPDFObjectHandle** may be constructed.

The *QPDF::resolve()* method, which is used to resolve an indirect object, may be invoked from the **QPDFObjectHandle** class. It first checks a cache to see whether this object has already been read. If not, it reads the object from the PDF file and caches it. It the returns the resulting **QPDFObjectHandle**. The calling object handle then replaces its **PointerHolder<QDFObject>** with the one from the newly returned **QPDFObjectHandle**. In this way, only a single copy of any direct object need exist and clients can access objects transparently without knowing caring whether they are direct or indirect objects. Additionally, no object is ever read from the file more than once. That means that only the portions of the PDF file that are actually needed are ever read from the input file, thus allowing the qpdf package to take advantage of this important design goal of PDF files.

If the requested object is inside of an object stream, the object stream itself is first read into memory. Then the tokenizer reads objects from the memory stream based on the offset information stored in the stream. Those individual objects are cached, after which the temporary buffer holding the object stream contents are discarded. In this way, the first time an object in an object stream is requested, all objects in the stream are cached.

An instance of **QPDF** is constructed by using the class's default constructor. If desired, the **QPDF** object may be configured with various methods that change its default behavior. Then the *QPDF::processFile()* method is passed the name of a PDF file, which permanently associates the file with that QPDF object. A password may also be given for access to password-protected files. QPDF does not enforce encryption parameters and will treat user and owner passwords equivalently. Either password may be used to access an encrypted file. [1] **QPDF** will allow recovery of a user password given an owner password. The input PDF file must be seekable. (Output files written by **QPDFWriter** need not be seekable, even when creating linearized files.) During construction, **QPDF** validates the PDF file's header, and then reads the cross reference tables and trailer dictionaries. The **QPDF** class keeps only the first trailer dictionary though it does read all of them so it can check the /Prev key. **QPDF** class users may request the root object and the trailer dictionary specifically. The cross reference table is kept private. Objects may then be requested by number of by walking the object tree.

When a PDF file has a cross-reference stream instead of a cross-reference table and trailer, requesting the document's trailer dictionary returns the stream dictionary from the cross-reference stream instead.

There are some convenience routines for very common operations such as walking the page tree and returning a vector of all page objects. For full details, please see the header file *QPDF.hh*.

The following example should clarify how **QPDF** processes a simple file.

- Client constructs **QPDF** *pdf* and calls *pdf.processFile("a.pdf");*.

- The **QPDF** class checks the beginning of *a.pdf* for `%!PDF-1.[0-9]+`. It then reads the cross reference table mentioned at the end of the file, ensuring that it is looking before the last `%%EOF`. After getting to `trailer` keyword, it invokes the parser.

---

[1] As pointed out earlier, the intention is not for qpdf to be used to bypass security on files. but as any open source PDF consumer may be easily modified to bypass basic PDF document security, and qpdf offers may transformations that can do this as well, there seems to be little point in the added complexity of conditionally enforcing document security.

- The parser sees "<<", so it calls itself recursively in dictionary creation mode.

- In dictionary creation mode, the parser keeps accumulating objects until it encounters ">>". Each object that is read is pushed onto a stack. If "R" is read, the last two objects on the stack are inspected. If they are integers, they are popped off the stack and their values are used to construct an indirect object handle which is then pushed onto the stack. When ">>" is finally read, the stack is converted into a **QPDF_Dictionary** which is placed in a **QPDFObjectHandle** and returned.

- The resulting dictionary is saved as the trailer dictionary.

- The /Prev key is searched. If present, **QPDF** seeks to that point and repeats except that the new trailer dictionary is not saved. If /Prev is not present, the initial parsing process is complete.

  If there is an encryption dictionary, the document's encryption parameters are initialized.

- The client requests root object. The **QPDF** class gets the value of root key from trailer dictionary and returns it. It is an unresolved indirect **QPDFObjectHandle**.

- The client requests the /Pages key from root **QPDFObjectHandle**. The **QPDFObjectHandle** notices that it is indirect so it asks **QPDF** to resolve it. **QPDF** looks in the object cache for an object with the root dictionary's object ID and generation number. Upon not seeing it, it checks the cross reference table, gets the offset, and reads the object present at that offset. It stores the result in the object cache and returns the cached result. The calling **QPDFObjectHandle** replaces its object pointer with the one from the resolved **QPDFObjectHandle**, verifies that it a valid dictionary object, and returns the (unresolved indirect) **QPDFObject** handle to the top of the Pages hierarchy.

  As the client continues to request objects, the same process is followed for each new requested object.

6.3. Casting Policy

This section describes the casting policy followed by qpdf's implementation. This is no concern to qpdf's end users and largely of no concern to people writing code that uses qpdf, but it could be of interest to people who are porting qpdf to a new platform or who are making modifications to the code.

The C++ code in qpdf is free of old-style casts except where unavoidable (e.g. where the old-style cast is in a macro provided by a third-party header file). When there is a need for a cast, it is handled, in order of preference, by rewriting the code to avoid the need for a cast, calling *const_cast*, calling *static_cast*, calling *reinterpret_cast*, or calling some combination of the above. As a last resort, a compiler-specific #pragma may be used to suppress a warning that we don't want to fix. Examples may include suppressing warnings about the use of old-style casts in code that is shared between C and C++ code.

The casting policy explicitly prohibits casting between integer sizes for no purpose other than to quiet a compiler warning when there is no reasonable chance of a problem resulting. The reason for this exclusion is that the practice of adding these additional casts precludes future use of additional compiler warnings as a tool for making future improvements to this aspect of the code, and it also damages the readability of the code.

There are a few significant areas where casting is common in the qpdf sources or where casting would be required to quiet higher levels of compiler warnings but is omitted at present:

- char vs. unsigned char. For historical reasons, there are a lot of places in qpdf's internals that deal with unsigned char, which means that a lot of casting is required to interoperate with standard library calls and std::string. In retrospect, qpdf should have probably used regular (signed) char and char* everywhere and just cast to unsigned char when needed, but it's too late to make that change now. There are *reinterpret_cast* calls to go between char* and unsigned char*, and there are *static_cast* calls to go between char and unsigned char. These should always be safe.

- Non-const `unsigned char*` used in the `Pipeline` interface. The pipeline interface has a *write* call that uses `unsigned char*` without a `const` qualifier. The main reason for this is to support pipelines that make calls to third-party libraries, such as zlib, that don't include `const` in their interfaces. Unfortunately, there are many places in the code where it is desirable to have `const char*` with pipelines. None of the pipeline implementations in qpdf currently modify the data passed to write, and doing so would be counter to the intent of `Pipeline`, but there is nothing in the code to prevent this from being done. There are places in the code where *const_cast* is used to remove the const-ness of pointers going into `Pipelines`. This could theoretically be unsafe, but there is adequate testing to assert that it is safe and will remain safe in qpdf's code.

- `size_t` vs. `qpdf_offset_t`. This is pretty much unavoidable since sizes are unsigned types and offsets are signed types. Whenever it is necessary to seek by an amount given by a `size_t`, it becomes necessary to mix and match between `size_t` and `qpdf_offset_t`. Additionally, qpdf sometimes treats memory buffers like files (as with `BufferInputSource`, and those seek interfaces have to be consistent with file-based input sources. Neither gcc nor MSVC give warnings for this case by default, but both have warning flags that can enable this. (MSVC: **/W14267** or **/W3**, which also enables some additional warnings that we ignore; gcc: **-Wconversion -Wsign-conversion**). This could matter for files whose sizes are larger than 2^{63} bytes, but it is reasonable to expect that a world where such files are common would also have larger `size_t` and `qpdf_offset_t` types in it. On most 64-bit systems at the time of this writing (the release of version 4.1.0 of qpdf), both `size_t` and `qpdf_offset_t` are 64-bit integer types, while on many current 32-bit systems, `size_t` is a 32-bit type while `qpdf_offset_t` is a 64-bit type. I am not aware of any cases where 32-bit systems that have `size_t` smaller than `qpdf_offset_t` could run into problems. Although I can't conclusively rule out the possibility of such problems existing, I suspect any cases would be pretty contrived. In the event that someone should produce a file that qpdf can't handle because of what is suspected to be issues involving the handling of `size_t` vs. `qpdf_offset_t` (such files may behave properly on 64-bit systems but not on 32-bit systems because they have very large embedded files or streams, for example), the above mentioned warning flags could be enabled and all those implicit conversions could be carefully scrutinized. (I have already gone through that exercise once in adding support for files larger than 4 GB in size.) I continue to be committed to supporting large files on 32-bit systems, but I would not go to any lengths to support corner cases involving large embedded files or large streams that work on 64-bit systems but not on 32-bit systems because of `size_t` being too small. It is reasonable to assume that anyone working with such files would be using a 64-bit system anyway since many 32-bit applications would have similar difficulties.

- `size_t` vs. `int` or `long`. There are some cases where `size_t` and `int` or `long` or `size_t` and `unsigned int` or `unsigned long` are used interchangeably. These cases occur when working with very small amounts of memory, such as with the bit readers (where we're working with just a few bytes at a time), some cases of *strlen*, and a few other cases. I have scrutinized all of these cases and determined them to be safe, but there is no mechanism in the code to ensure that new unsafe conversions between `int` and `size_t` aren't introduced short of good testing and strong awareness of the issues. Again, if any such bugs are suspected in the future, enabling the additional warning flags and scrutinizing the warnings would be in order.

To be clear, I believe qpdf to be well-behaved with respect to sizes and offsets, and qpdf's test suite includes actual generation and full processing of files larger than 4 GB in size. The issues raised here are largely academic and should not in any way be interpreted to mean that qpdf has practical problems involving sloppiness with integer types. I also believe that appropriate measures have been taken in the code to avoid problems with signed vs. unsigned integers from resulting in memory overwrites or other issues with potential security implications, though there are never any absolute guarantees.

6.4. Encryption

Encryption is supported transparently by qpdf. When opening a PDF file, if an encryption dictionary exists, the **QPDF** object processes this dictionary using the password (if any) provided. The primary decryption key is computed and cached. No further access is made to the encryption dictionary after that time. When an object is read from a file, the object ID and generation of the object in which it is contained is always known. Using this information along with the stored encryption key, all stream and string objects are transparently decrypted. Raw encrypted objects are never stored in memory. This way, nothing in the library ever has to know or care whether it is reading an encrypted file.

An interface is also provided for writing encrypted streams and strings given an encryption key. This is used by **QPDFWriter** when it rewrites encrypted files.

When copying encrypted files, unless otherwise directed, qpdf will preserve any encryption in force in the original file. qpdf can do this with either the user or the owner password. There is no difference in capability based on which password is used. When 40 or 128 bit encryption keys are used, the user password can be recovered with the owner password. With 256 keys, the user and owner passwords are used independently to encrypt the actual encryption key, so while either can be used, the owner password can no longer be used to recover the user password.

Starting with version 4.0.0, qpdf can read files that are not encrypted but that contain encrypted attachments, but it cannot write such files. qpdf also requires the password to be specified in order to open the file, not just to extract attachments, since once the file is open, all decryption is handled transparently. When copying files like this while preserving encryption, qpdf will apply the file's encryption to everything in the file, not just to the attachments. When decrypting the file, qpdf will decrypt the attachments. In general, when copying PDF files with multiple encryption formats, qpdf will choose the newest format. The only exception to this is that clear-text metadata will be preserved as clear-text if it is that way in the original file.

6.5. Adding and Removing Pages

While qpdf's API has supported adding and modifying objects for some time, version 3.0 introduces specific methods for adding and removing pages. These are largely convenience routines that handle two tricky issues: pushing inheritable resources from the `/Pages` tree down to individual pages and manipulation of the `/Pages` tree itself. For details, see *addPage* and surrounding methods in *QPDF.hh*.

6.6. Reserving Object Numbers

Version 3.0 of qpdf introduced the concept of reserved objects. These are seldom needed for ordinary operations, but there are cases in which you may want to add a series of indirect objects with references to each other to a **QPDF** object. This causes a problem because you can't determine the object ID that a new indirect object will have until you add it to the **QPDF** object with *QPDF::makeIndirectObject*. The only way to add two mutually referential objects to a **QPDF** object prior to version 3.0 would be to add the new objects first and then make them refer to each other after adding them. Now it is possible to create a *reserved object* using *QPDFObjectHandle::newReserved*. This is an indirect object that stays "unresolved" even if it is queried for its type. So now, if you want to create a set of mutually referential objects, you can create reservations for each one of them and use those reservations to construct the references. When finished, you can call *QPDF::replaceReserved* to replace the reserved objects with the real ones. This functionality will never be needed by most applications, but it is used internally by QPDF when copying objects from other PDF files, as discussed in . For an example of how to use reserved objects, search for *newReserved* in *test_driver.cc* in qpdf's sources.

6.7. Copying Objects From Other PDF Files

Version 3.0 of qpdf introduced the ability to copy objects into a **QPDF** object from a different **QPDF** object, which we refer to as *foreign objects*. This allows arbitrary merging of PDF files. The **qpdf** command-line tool provides limited support for basic page selection, including merging in pages from other files, but the library's API makes it possible to implement arbitrarily complex merging operations. The main method for copying foreign objects is *QPDF::copyForeignObject*. This takes an indirect object from another **QPDF** and copies it recursively into this object while preserving all object structure, including circular references. This means you can add a direct object that you create from scratch to a **QPDF** object with *QPDF::makeIndirectObject*, and you can add an indirect object from another file with *QPDF::copyForeignObject*. The fact that *QPDF::makeIndirectObject* does not automatically detect a foreign object and copy it is an explicit design decision. Copying a foreign object seems like a sufficiently significant thing to do that it should be done explicitly.

The other way to copy foreign objects is by passing a page from one **QPDF** to another by calling *QPDF::addPage*. In contrast to *QPDF::makeIndirectObject*, this method automatically distinguishes between indirect objects in the current file, foreign objects, and direct objects.

6.8. Writing PDF Files

The qpdf library supports file writing of **QPDF** objects to PDF files through the **QPDFWriter** class. The **QPDFWriter** class has two writing modes: one for non-linearized files, and one for linearized files. See Chapter 7, *Linearization*, page 22 for a description of linearization is implemented. This section describes how we write non-linearized files including the creation of QDF files (see Chapter 4, *QDF Mode*, page 12.

This outline was written prior to implementation and is not exactly accurate, but it provides a correct "notional" idea of how writing works. Look at the code in **QPDFWriter** for exact details.

- Initialize state:

  - next object number = 1

  - object queue = empty

  - renumber table: old object id/generation to new id/0 = empty

  - xref table: new id -> offset = empty

- Create a QPDF object from a file.

- Write header for new PDF file.

- Request the trailer dictionary.

- For each value that is an indirect object, grab the next object number (via an operation that returns and increments the number). Map object to new number in renumber table. Push object onto queue.

- While there are more objects on the queue:

  - Pop queue.

  - Look up object's new number *n* in the renumbering table.

  - Store current offset into xref table.

  - Write `n 0 obj`.

  - If object is null, whether direct or indirect, write out null, thus eliminating unresolvable indirect object references.

  - If the object is a stream stream, write stream contents, piped through any filters as required, to a memory buffer. Use this buffer to determine the stream length.

  - If object is not a stream, array, or dictionary, write out its contents.

  - If object is an array or dictionary (including stream), traverse its elements (for array) or values (for dictionaries), handling recursive dictionaries and arrays, looking for indirect objects. When an indirect object is found, if it is not resolvable, ignore. (This case is handled when writing it out.) Otherwise, look it up in the renumbering table. If not found, grab the next available object number, assign to the referenced object in the renumbering table, and push the referenced object onto the queue. As a special case, when writing out a stream dictionary, replace length, filters, and decode parameters as required.

Write out dictionary or array, replacing any unresolvable indirect object references with null (pdf spec says reference to non-existent object is legal and resolves to null) and any resolvable ones with references to the renumbered objects.

- If the object is a stream, write `stream\n`, the stream contents (from the memory buffer), and `\nendstream\n`.

- When done, write `endobj`.

Once we have finished the queue, all referenced objects will have been written out and all deleted objects or unreferenced objects will have been skipped. The new cross-reference table will contain an offset for every new object number from 1 up to the number of objects written. This can be used to write out a new xref table. Finally we can write out the trailer dictionary with appropriately computed /ID (see spec, 8.3, File Identifiers), the cross reference table offset, and `%%EOF`.

6.9. Filtered Streams

Support for streams is implemented through the **Pipeline** interface which was designed for this package.

When reading streams, create a series of **Pipeline** objects. The **Pipeline** abstract base requires implementation *write()* and *finish()* and provides an implementation of *getNext()*. Each pipeline object, upon receiving data, does whatever it is going to do and then writes the data (possibly modified) to its successor. Alternatively, a pipeline may be an end-of-the-line pipeline that does something like store its output to a file or a memory buffer ignoring a successor. For additional details, look at *Pipeline.hh*.

QPDF can read raw or filtered streams. When reading a filtered stream, the **QPDF** class creates a **Pipeline** object for one of each appropriate filter object and chains them together. The last filter should write to whatever type of output is required. The **QPDF** class has an interface to write raw or filtered stream contents to a given pipeline.

Chapter 7. Linearization

This chapter describes how **QPDF** and **QPDFWriter** implement creation and processing of linearized PDFS.

7.1. Basic Strategy for Linearization

To avoid the incestuous problem of having the qpdf library validate its own linearized files, we have a special linearized file checking mode which can be invoked via **qpdf --check-linearization** (or **qpdf --check**). This mode reads the linearization parameter dictionary and the hint streams and validates that object ordering, parameters, and hint stream contents are correct. The validation code was first tested against linearized files created by external tools (Acrobat and pdlin) and then used to validate files created by **QPDFWriter** itself.

7.2. Preparing For Linearization

Before creating a linearized PDF file from any other PDF file, the PDF file must be altered such that all page attributes are propagated down to the page level (and not inherited from parents in the `/Pages` tree). We also have to know which objects refer to which other objects, being concerned with page boundaries and a few other cases. We refer to this part of preparing the PDF file as *optimization*, discussed in Section 7.3, "Optimization", page 22 Note the, in this context, the term *optimization* is a qpdf term, and the term *linearization* is a term from the PDF specification. Do not be confused by the fact that many applications refer to linearization as optimization or web optimization.

When creating linearized PDF files from optimized PDF files, there are really only a few issues that need to be dealt with:

- Creation of hints tables

- Placing objects in the correct order

- Filling in offsets and byte sizes

7.3. Optimization

In order to perform various operations such as linearization and splitting files into pages, it is necessary to know which objects are referenced by which pages, page thumbnails, and root and trailer dictionary keys. It is also necessary to ensure that all page-level attributes appear directly at the page level and are not inherited from parents in the pages tree.

We refer to the process of enforcing these constraints as *optimization*. As mentioned above, note that some applications refer to linearization as optimization. Although this optimization was initially motivated by the need to create linearized files, we are using these terms separately.

PDF file optimization is implemented in the *QPDF_optimization.cc* source file. That file is richly commented and serves as the primary reference for the optimization process.

After optimization has been completed, the private member variables *obj_user_to_objects* and *object_to_obj_users* in **QPDF** have been populated. Any object that has more than one value in the *object_to_obj_users* table is shared. Any object that has exactly one value in the *object_to_obj_users* table is private. To find all the private objects in a page or a trailer or root dictionary key, one merely has make this determination for each element in the *obj_user_to_objects* table for the given page or key.

Note that pages and thumbnails have different object user types, so the above test on a page will not include objects referenced by the page's thumbnail dictionary and nothing else.

7.4. Writing Linearized Files

We will create files with only primary hint streams. We will never write overflow hint streams. (As of PDF version 1.4, Acrobat doesn't either, and they are never necessary.) The hint streams contain offset information to objects that point to where they would be if the hint stream were not present. This means that we have to calculate all object positions before we can generate and write the hint table. This means that we have to generate the file in two passes. To make this reliable, **QPDFWriter** in linearization mode invokes exactly the same code twice to write the file to a pipeline.

In the first pass, the target pipeline is a count pipeline chained to a discard pipeline. The count pipeline simply passes its data through to the next pipeline in the chain but can return the number of bytes passed through it at any intermediate point. The discard pipeline is an end of line pipeline that just throws its data away. The hint stream is not written and dummy values with adequate padding are stored in the first cross reference table, linearization parameter dictionary, and /Prev key of the first trailer dictionary. All the offset, length, object renumbering information, and anything else we need for the second pass is stored.

At the end of the first pass, this information is passed to the **QPDF** class which constructs a compressed hint stream in a memory buffer and returns it. **QPDFWriter** uses this information to write a complete hint stream object into a memory buffer. At this point, the length of the hint stream is known.

In the second pass, the end of the pipeline chain is a regular file instead of a discard pipeline, and we have known values for all the offsets and lengths that we didn't have in the first pass. We have to adjust offsets that appear after the start of the hint stream by the length of the hint stream, which is known. Anything that is of variable length is padded, with the padding code surrounding any writing code that differs in the two passes. This ensures that changes to the way things are represented never results in offsets that were gathered during the first pass becoming incorrect for the second pass.

Using this strategy, we can write linearized files to a non-seekable output stream with only a single pass to disk or wherever the output is going.

7.5. Calculating Linearization Data

Once a file is optimized, we have information about which objects access which other objects. We can then process these tables to decide which part (as described in "Linearized PDF Document Structure" in the PDF specification) each object is contained within. This tells us the exact order in which objects are written. The **QPDFWriter** class asks for this information and enqueues objects for writing in the proper order. It also turns on a check that causes an exception to be thrown if an object is encountered that has not already been queued. (This could happen only if there were a bug in the traversal code used to calculate the linearization data.)

7.6. Known Issues with Linearization

There are a handful of known issues with this linearization code. These issues do not appear to impact the behavior of linearized files which still work as intended: it is possible for a web browser to begin to display them before they are fully downloaded. In fact, it seems that various other programs that create linearized files have many of these same issues. These items make reference to terminology used in the linearization appendix of the PDF specification.

- Thread Dictionary information keys appear in part 4 with the rest of Threads instead of in part 9. Objects in part 9 are not grouped together functionally.

- We are not calculating numerators for shared object positions within content streams or interleaving them within content streams.

- We generate only page offset, shared object, and outline hint tables. It would be relatively easy to add some additional tables. We gather most of the information needed to create thumbnail hint tables. There are comments in the code about this.

7.7. Debugging Note

The **qpdf --show-linearization** command can show the complete contents of linearization hint streams. To look at the raw data, you can extract the filtered contents of the linearization hint tables using **qpdf --show-object=n --filtered-stream-data**. Then, to convert this into a bit stream (since linearization tables are bit streams written without regard to byte boundaries), you can pipe the resulting data through the following perl code:

```
use bytes;
binmode STDIN;
undef $/;
my $a = <STDIN>;
my @ch = split(//, $a);
map { printf("%08b", ord($_)) } @ch;
print "\n";
```

Chapter 8. Object and Cross-Reference Streams

This chapter provides information about the implementation of object stream and cross-reference stream support in qpdf.

8.1. Object Streams

Object streams can contain any regular object except the following:

- stream objects

- objects with generation > 0

- the encryption dictionary

- objects containing the /Length of another stream

In addition, Adobe reader (at least as of version 8.0.0) appears to not be able to handle having the document catalog appear in an object stream if the file is encrypted, though this is not specifically disallowed by the specification.

There are additional restrictions for linearized files. See Section 8.3, "Implications for Linearized Files," page 26 for details.

The PDF specification refers to objects in object streams as "compressed objects" regardless of whether the object stream is compressed.

The generation number of every object in an object stream must be zero. It is possible to delete and replace an object in an object stream with a regular object.

The object stream dictionary has the following keys:

- `/N`: number of objects

- `/First`: byte offset of first object

- `/Extends`: indirect reference to stream that this extends

Stream collections are formed with `/Extends`. They must form a directed acyclic graph. These can be used for semantic information and are not meaningful to the PDF document's syntactic structure. Although qpdf preserves stream collections, it never generates them and doesn't make use of this information in any way.

The specification recommends limiting the number of objects in object stream for efficiency in reading and decoding. Acrobat 6 uses no more than 100 objects per object stream for linearized files and no more 200 objects per stream for non-linearized files. **QPDFWriter**, in object stream generation mode, never puts more than 100 objects in an object stream.

Object stream contents consists of *N* pairs of integers, each of which is the object number and the byte offset of the object relative to the first object in the stream, followed by the objects themselves, concatenated.

8.2. Cross-Reference Streams

For non-hybrid files, the value following `startxref` is the byte offset to the xref stream rather than the word `xref`.

For hybrid files (files containing both xref tables and cross-reference streams), the xref table's trailer dictionary contains the key /XRefStm whose value is the byte offset to a cross-reference stream that supplements the xref table. A PDF 1.5-compliant application should read the xref table first. Then it should replace any object that it has already seen with any defined in the xref stream. Then it should follow any /Prev pointer in the original xref table's trailer dictionary. The specification is not clear about what should be done, if anything, with a /Prev pointer in the xref stream referenced by an xref table. The **QPDF** class ignores it, which is probably reasonable since, if this case were to appear for any sensible PDF file, the previous xref table would probably have a corresponding /XRefStm pointer of its own. For example, if a hybrid file were appended, the appended section would have its own xref table and /XRefStm. The appended xref table would point to the previous xref table which would point the /XRefStm, meaning that the new /XRefStm doesn't have to point to it.

Since xref streams must be read very early, they may not be encrypted, and the may not contain indirect objects for keys required to read them, which are these:

- /Type: value /XRef

- /Size: value *n+1*: where *n* is highest object number (same as /Size in the trailer dictionary)

- /Index (optional): value [*n count* ...] used to determine which objects' information is stored in this stream. The default is [0 /Size].

- /Prev: value *offset*: byte offset of previous xref stream (same as /Prev in the trailer dictionary)

- /W [...]: sizes of each field in the xref table

The other fields in the xref stream, which may be indirect if desired, are the union of those from the xref table's trailer dictionary.

8.2.1. Cross-Reference Stream Data

The stream data is binary and encoded in big-endian byte order. Entries are concatenated, and each entry has a length equal to the total of the entries in /W above. Each entry consists of one or more fields, the first of which is the type of the field. The number of bytes for each field is given by /W above. A 0 in /W indicates that the field is omitted and has the default value. The default value for the field type is "1". All other default values are "0".

PDF 1.5 has three field types:

- 0: for free objects. Format: 0 obj next-generation, same as the free table in a traditional cross-reference table

- 1: regular non-compressed object. Format: 1 offset generation

- 2: for objects in object streams. Format: 2 object-stream-number index, the number of object stream containing the object and the index within the object stream of the object.

It seems standard to have the first entry in the table be 0 0 0 instead of 0 0 ffff if there are no deleted objects.

8.3. Implications for Linearized Files

For linearized files, the linearization dictionary, document catalog, and page objects may not be contained in object streams.

Objects stored within object streams are given the highest range of object numbers within the main and first-page cross-reference sections.

It is okay to use cross-reference streams in place of regular xref tables. There are on special considerations.

Hint data refers to object streams themselves, not the objects in the streams. Shared object references should also be made to the object streams. There are no reference in any hint tables to the object numbers of compressed objects (objects within object streams).

When numbering objects, all shared objects within both the first and second halves of the linearized files must be numbered consecutively after all normal uncompressed objects in that half.

8.4. Implementation Notes

There are three modes for writing object streams: **disable**, **preserve**, and **generate**. In disable mode, we do not generate any object streams, and we also generate an xref table rather than xref streams. This can be used to generate PDF files that are viewable with older readers. In preserve mode, we write object streams such that written object streams contain the same objects and `/Extends` relationships as in the original file. This is equal to disable if the file has no object streams. In generate, we create object streams ourselves by grouping objects that are allowed in object streams together in sets of no more than 100 objects. We also ensure that the PDF version is at least 1.5 in generate mode, but we preserve the version header in the other modes. The default is **preserve**.

We do not support creation of hybrid files. When we write files, even in preserve mode, we will lose any xref tables and merge any appended sections.

Appendix A. Release Notes

For a detailed list of changes, please see the file *ChangeLog* in the source distribution.

4.1.0: April 14, 2013

- Note to people including qpdf in distributions: the *.la* files generated by libtool are now installed by qpdf's **make install** target. Before, they were not installed. This means that if your distribution does not want to include *.la* files, you must remove them as part of your packaging process.

- Major enhancement: API enhancements have been made to support parsing of content streams. This enhancement includes the following changes:

  - *QPDFObjectHandle::parseContentStream* method parses objects in a content stream and calls handlers in a callback class. The example *examples/pdf-parse-content.cc* illustrates how this may be used.

  - QPDFObjectHandle can now represent operators and inline images, object types that may only appear in content streams.

  - Method *QPDFObjectHandle::getTypeCode()* returns an enumerated type value representing the underlying object type. Method *QPDFObjectHandle::getTypeName()* returns a text string describing the name of the type of a QPDFObjectHandle object. These methods can be used for more efficient parsing and debugging/diagnostic messages.

- **qpdf --check** now parses all pages' content streams in addition to doing other checks. While there are still many types of errors that cannot be detected, syntactic errors in content streams will now be reported.

- Minor compilation enhancements have been made to facilitate easier for support for a broader range of compilers and compiler versions.

  - Warning flags have been moved into a separate variable in *autoconf.mk*

  - The configure flag **--enable-werror** work for Microsoft compilers

  - All MSVC CRT security warnings have been resolved.

  - All C-style casts in C++ Code have been replaced by C++ casts, and many casts that had been included to suppress higher warning levels for some compilers have been removed, primarily for clarity. Places where integer type coercion occurs have been scrutinized. A new casting policy has been documented in the manual. This is of concern mainly to people porting qpdf to new platforms or compilers. It is not visible to programmers writing code that uses the library

  - Some internal limits have been removed in code that converts numbers to strings. This is largely invisible to users, but it does trigger a bug in some older versions of mingw-w64's C++ library. See *README-windows.txt* in the source distribution if you think this may affect you. The copy of the DLL distributed with qpdf's binary distribution is not affected by this problem.

- The RPM spec file previously included with qpdf has been removed. This is because virtually all Linux distributions include qpdf now that it is a dependency of CUPS filters.

- A few bug fixes are included:

  - Overridden compressed objects are properly handled. Before, there were certain constructs that could cause qpdf to see old versions of some objects. The most usual manifestation of this was loss of filled in form values for certain files.

- Installation no longer uses GNU/Linux-specific versions of some commands, so **make install** works on Solaris with native tools.

- The 64-bit mingw Windows binary package no longer includes a 32-bit DLL.

4.0.1: January 17, 2013

- Fix detection of binary attachments in test suite to avoid false test failures on some platforms.

- Add clarifying comment in *QPDF.hh* to methods that return the user password explaining that it is no longer possible with newer encryption formats to recover the user password knowing the owner password. In earlier encryption formats, the user password was encrypted in the file using the owner password. In newer encryption formats, a separate encryption key is used on the file, and that key is independently encrypted using both the user password and the owner password.

4.0.0: December 31, 2012

- Major enhancement: support has been added for newer encryption schemes supported by version X of Adobe Acrobat. This includes use of 127-character passwords, 256-bit encryption keys, and the encryption scheme specified in ISO 32000-2, the PDF 2.0 specification. This scheme can be chosen from the command line by specifying use of 256-bit keys. qpdf also supports the deprecated encryption method used by Acrobat IX. This encryption style has known security weaknesses and should not be used in practice. However, such files exist "in the wild," so support for this scheme is still useful. New methods *QPDFWriter::setR6EncryptionParameters* (for the PDF 2.0 scheme) and *QPDFWriter::setR5EncryptionParameters* (for the deprecated scheme) have been added to enable these new encryption schemes. Corresponding functions have been added to the C API as well.

- Full support for Adobe extension levels in PDF version information. Starting with PDF version 1.7, corresponding to ISO 32000, Adobe adds new functionality by increasing the extension level rather than increasing the version. This support includes addition of the *QPDF::getExtensionLevel* method for retrieving the document's extension level, addition of versions of *QPDFWriter::setMinimumPDFVersion* and *QPDFWriter::forcePDFVersion* that accept an extension level, and extended syntax for specifying forced and minimum versions on the command line as described in Section 3.5, "Advanced Transformation Options", page 8. Corresponding functions have been added to the C API as well.

- Minor fixes to prevent qpdf from referencing objects in the file that are not referenced in the file's overall structure. Most files don't have any such objects, but some files have contain unreferenced objects with errors, so these fixes prevent qpdf from needlessly rejecting or complaining about such objects.

- Add new generalized methods for reading and writing files from/to programmer-defined sources. The method *QPDF::processInputSource* allows the programmer to use any input source for the input file, and *QPDFWriter::setOutputPipeline* allows the programmer to write the output file through any pipeline. These methods would make it possible to perform any number of specialized operations, such as accessing external storage systems, creating bindings for qpdf in other programming languages that have their own I/O systems, etc.

- Add new method *QPDF::getEncryptionKey* for retrieving the underlying encryption key used in the file.

- This release includes a small handful of non-compatible API changes. While effort is made to avoid such changes, all the non-compatible API changes in this version were to parts of the API that would likely never be used outside the library itself. In all cases, the altered methods or structures were parts of the **QPDF** that were public to enable them to be called from either **QPDFWriter** or were part of validation code that was over-zealous in reporting problems in parts of the file that would not ordinarily be referenced. In no case did any of the removed methods do anything worse that falsely report error conditions in files that were broken in ways that didn't matter. The following public parts of the **QPDF** class were changed in a non-compatible way:

- Updated nested **QPDF::EncryptionData** class to add fields needed by the newer encryption formats, member variables changed to private so that future changes will not require breaking backward compatibility.

- Added additional parameters to *compute_data_key*, which is used by **QPDFWriter** to compute the encryption key used to encrypt a specific object.

- Removed the method *flattenScalarReferences*. This method was previously used prior to writing a new PDF file, but it has the undesired side effect of causing qpdf to read objects in the file that were not referenced. Some otherwise files have unreferenced objects with errors in them, so this could cause qpdf to reject files that would be accepted by virtually all other PDF readers. In fact, qpdf relied on only a very small part of what flattenScalarReferences did, so only this part has been preserved, and it is now done directly inside **QPDFWriter**.

- Removed the method *decodeStreams*. This method was used by the **--check** option of the **qpdf** command-line tool to force all streams in the file to be decoded, but it also suffered from the problem of opening otherwise unreferenced streams and thus could report false positive. The **--check** option now causes qpdf to go through all the motions of writing a new file based on the original one, so it will always reference and check exactly those parts of a file that any ordinary viewer would check.

- Removed the method *trimTrailerForWrite*. This method was used by **QPDFWriter** to modify the original QPDF object by removing fields from the trailer dictionary that wouldn't apply to the newly written file. This functionality, though generally harmless, was a poor implementation and has been replaced by having QPDFWriter filter these out when copying the trailer rather than modifying the original QPDF object. (Note that qpdf never modifies the original file itself.)

- Allow the PDF header to appear anywhere in the first 1024 bytes of the file. This is consistent with what other readers do.

- Fix the **pkg-config** files to list zlib and pcre in *Requires.private* to better support static linking using **pkg-config**.

3.0.2: September 6, 2012

- Bug fix: *QPDFWriter::setOutputMemory* did not work when not used with *QPDFWriter::setStaticID*, which made it pretty much useless. This has been fixed.

- New API call *QPDFWriter::setExtraHeaderText* inserts additional text near the header of the PDF file. The intended use case is to insert comments that may be consumed by a downstream application, though other use cases may exist.

3.0.1: August 11, 2012

- Version 3.0.0 included addition of files for **pkg-config**, but this was not mentioned in the release notes. The release notes for 3.0.0 were updated to mention this.

- Bug fix: if an object stream ended with a scalar object not followed by space, qpdf would incorrectly report that it encountered a premature EOF. This bug has been in qpdf since version 2.0.

3.0.0: August 2, 2012

- Acknowledgment: I would like to express gratitude for the contributions of Tobias Hoffmann toward the release of qpdf version 3.0. He is responsible for most of the implementation and design of the new API for manipulating pages, and contributed code and ideas for many of the improvements made in version 3.0. Without his work, this release would certainly not have happened as soon as it did, if at all.

- *Non-compatible API change:* The version of *QPDFObjectHandle::replaceStreamData* that uses a **Stream-DataProvider** no longer requires (or accepts) a *length* parameter. See Appendix C, *Upgrading to 3.0,* page

37 for an explanation. While care is taken to avoid non-compatible API changes in general, an exception was made this time because the new interface offers an opportunity to significantly simplify calling code.

- Support has been added for large files. The test suite verifies support for files larger than 4 gigabytes, and manual testing has verified support for files larger than 10 gigabytes. Large file support is available for both 32-bit and 64-bit platforms as long as the compiler and underlying platforms support it.

- Support for page selection (splitting and merging PDF files) has been added to the **qpdf** command-line tool. See Section 3.4, "Page Selection Options", page 6.

- Options have been added to the **qpdf** command-line tool for copying encryption parameters from another file. See Section 3.2, "Basic Options", page 4.

- New methods have been added to the *QPDF* object for adding and removing pages. See Section 6.5, "Adding and Removing Pages", page 19.

- New methods have been added to the *QPDF* object for copying objects from other PDF files. See Section 6.7, "Copying Objects From Other PDF Files", page 19

- A new method *QPDFObjectHandle::parse* has been added for constructing *QPDFObjectHandle* objects from a string description.

- Methods have been added to *QPDFWriter* to allow writing to an already open stdio `FILE*` addition to writing to standard output or a named file. Methods have been added to *QPDF* to be able to process a file from an already open stdio `FILE*`. This makes it possible to read and write PDF from secure temporary files that have been unlinked prior to being fully read or written.

- The *QPDF::emptyPDF* can be used to allow creation of PDF files from scratch. The example *examples/pdf-create.cc* illustrates how it can be used.

- Several methods to take *PointerHolder<Buffer>* can now also accept `std::string` arguments.

- Many new convenience methods have been added to the library, most in *QPDFObjectHandle*. See *ChangeLog* for a full list.

- When building on a platform that supports ELF shared libraries (such as Linux), symbol versions are enabled by default. They can be disabled by passing **--disable-ld-version-script** to **./configure**.

- The file *libqpdf.pc* is now installed to support **pkg-config**.

- Image comparison tests are off by default now since they are not needed to verify a correct build or port of qpdf. They are needed only when changing the actual PDF output generated by qpdf. You should enable them if you are making deep changes to qpdf itself. See *README* for details.

- Large file tests are off by default but can be turned on with **./configure** or by setting an environment variable before running the test suite. See *README* for details.

- When qpdf's test suite fails, failures are not printed to the terminal anymore by default. Instead, find them in *build/qtest.log*. For packagers who are building with an autobuilder, you can add the **--enable-show-failed-test-output** option to **./configure** to restore the old behavior.

2.3.1: December 28, 2011

- Fix thread-safety problem resulting from non-thread-safe use of the PCRE library.

- Made a few minor documentation fixes.

- Add workaround for a bug that appears in some versions of ghostscript to the test suite

- Fix minor build issue for Visual C++ 2010.

2.3.0: August 11, 2011

- Bug fix: when preserving existing encryption on encrypted files with cleartext metadata, older qpdf versions would generate password-protected files with no valid password. This operation now works. This bug only affected files created by copying existing encryption parameters; explicit encryption with specification of cleartext metadata worked before and continues to work.

- Enhance **QPDFWriter** with a new constructor that allows you to delay the specification of the output file. When using this constructor, you may now call *QPDFWriter::setOutputFilename* to specify the output file, or you may use *QPDFWriter::setOutputMemory* to cause **QPDFWriter** to write the resulting PDF file to a memory buffer. You may then use *QPDFWriter::getBuffer* to retrieve the memory buffer.

- Add new API call *QPDF::replaceObject* for replacing objects by object ID

- Add new API call *QPDF::swapObjects* for swapping two objects by object ID

- Add *QPDFObjectHandle::getDictAsMap* and *QPDFObjectHandle::getArrayAsVector* to allow retrieval of dictionary objects as maps and array objects as vectors.

- Add functions *qpdf_get_info_key* and *qpdf_set_info_key* to the C API for manipulating string fields of the document's /Info dictionary.

- Add functions *qpdf_init_write_memory*, *qpdf_get_buffer_length*, and *qpdf_get_buffer* to the C API for writing PDF files to a memory buffer instead of a file.

2.2.4: June 25, 2011

- Fix installation and compilation issues; no functionality changes.

2.2.3: April 30, 2011

- Handle some damaged streams with incorrect characters following the stream keyword.

- Improve handling of inline images when normalizing content streams.

- Enhance error recovery to properly handle files that use object 0 as a regular object, which is specifically disallowed by the spec.

2.2.2: October 4, 2010

- Add new function *qpdf_read_memory* to the C API to call *QPDF::processMemoryFile*. This was an omission in qpdf 2.2.1.

2.2.1: October 1, 2010

- Add new method *QPDF::setOutputStreams* to replace *std::cout* and *std::cerr* with other streams for generation of diagnostic messages and error messages. This can be useful for GUIs or other applications that want to capture any output generated by the library to present to the user in some other way. Note that QPDF does not write to *std::cout* (or the specified output stream) except where explicitly mentioned in *QPDF.hh*, and that the only use of the error stream is for warnings. Note also that output of warnings is suppressed when setSuppressWarnings(true) is called.

- Add new method *QPDF::processMemoryFile* for operating on PDF files that are loaded into memory rather than in a file on disk.

- Give a warning but otherwise ignore empty PDF objects by treating them as null. Empty object are not permitted by the PDF specification but have been known to appear in some actual PDF files.

- Handle inline image filter abbreviations when the appear as stream filter abbreviations. The PDF specification does not allow use of stream filter abbreviations in this way, but Adobe Reader and some other PDF readers accept them since they sometimes appear incorrectly in actual PDF files.

- Implement miscellaneous enhancements to **PointerHolder** and **Buffer** to support other changes.

2.2.0: August 14, 2010

- Add new methods to **QPDFObjectHandle** (*newStream* and *replaceStreamData* for creating new streams and replacing stream data. This makes it possible to perform a wide range of operations that were not previously possible.

- Add new helper method in **QPDFObjectHandle** (*addPageContents*) for appending or prepending new content streams to a page. This method makes it possible to manipulate content streams without having to be concerned whether a page's contents are a single stream or an array of streams.

- Add new method in **QPDFObjectHandle**: *replaceOrRemoveKey*, which replaces a dictionary key with a given value unless the value is null, in which case it removes the key instead.

- Add new method in **QPDFObjectHandle**: *getRawStreamData*, which returns the raw (unfiltered) stream data into a buffer. This complements the *getStreamData* method, which returns the filtered (uncompressed) stream data and can only be used when the stream's data is filterable.

- Provide two new examples: **pdf-double-page-size** and **pdf-invert-images** that illustrate the newly added interfaces.

- Fix a memory leak that would cause loss of a few bytes for every object involved in a cycle of object references. Thanks to Jian Ma for calling my attention to the leak.

2.1.5: April 25, 2010

- Remove restriction of file identifier strings to 16 bytes. This unnecessary restriction was preventing qpdf from being able to encrypt or decrypt files with identifier strings that were not exactly 16 bytes long. The specification imposes no such restriction.

2.1.4: April 18, 2010

- Apply the same padding calculation fix from version 2.1.2 to the main cross reference stream as well.

- Since **qpdf --check** only performs limited checks, clarify the output to make it clear that there still may be errors that qpdf can't check. This should make it less surprising to people when another PDF reader is unable to read a file that qpdf thinks is okay.

2.1.3: March 27, 2010

- Fix bug that could cause a failure when rewriting PDF files that contain object streams with unreferenced objects that in turn reference indirect scalars.

- Don't complain about (invalid) AES streams that aren't a multiple of 16 bytes. Instead, pad them before decrypting.

2.1.2: January 24, 2010

- Fix bug in padding around first half cross reference stream in linearized files. The bug could cause an assertion failure when linearizing certain unlucky files.

2.1.1: December 14, 2009

- No changes in functionality; insert missing include in an internal library header file to support gcc 4.4, and update test suite to ignore broken Adobe Reader installations.

2.1: October 30, 2009

- This is the first version of qpdf to include Windows support. On Windows, it is possible to build a DLL. Additionally, a partial C-language API has been introduced, which makes it possible to call qpdf functions from non-C++ environments. I am very grateful to Zarko Gagic (http://delphi.about.com/) for tirelessly testing numerous pre-release versions of this DLL and providing many excellent suggestions on improving the interface.

  For programming to the C interface, please see the header file *qpdf/qpdf-c.h* and the example *examples/pdf-linearize.c*.

- Zarko Gajic has written a Delphi wrapper for qpdf, which can be downloaded from qpdf's download side. Zarko's Delphi wrapper is released with the same licensing terms as qpdf itself and comes with this disclaimer: "Delphi wrapper unit *qpdf.pas* created by Zarko Gajic (http://delphi.about.com/). Use at your own risk and for whatever purpose you want. No support is provided. Sample code is provided."

- Support has been added for AES encryption and crypt filters. Although qpdf does not presently support files that use PKI-based encryption, with the addition of AES and crypt filters, qpdf is now be able to open most encrypted files created with newer versions of Acrobat or other PDF creation software. Note that I have not been able to get very many files encrypted in this way, so it's possible there could still be some cases that qpdf can't handle. Please report them if you find them.

- Many error messages have been improved to include more information in hopes of making qpdf a more useful tool for PDF experts to use in manually recovering damaged PDF files.

- Attempt to avoid compressing metadata streams if possible. This is consistent with other PDF creation applications.

- Provide new command-line options for AES encrypt, cleartext metadata, and setting the minimum and forced PDF versions of output files.

- Add additional methods to the **QPDF** object for querying the document's permissions. Although qpdf does not enforce these permissions, it does make them available so that applications that use qpdf can enforce permissions.

- The **--check** option to **qpdf** has been extended to include some additional information.

- There have been a handful of non-compatible API changes. For details, see Appendix B, *Upgrading from 2.0 to 2.1*, page 36.

2.0.6: May 3, 2009

- Do not attempt to uncompress streams that have decode parameters we don't recognize. Earlier versions of qpdf would have rejected files with such streams.

2.0.5: March 10, 2009

- Improve error handling in the LZW decoder, and fix a small error introduced in the previous version with regard to handling full tables. The LZW decoder has been more strongly verified in this release.

2.0.4: February 21, 2009

- Include proper support for LZW streams encoded without the "early code change" flag. Special thanks to Atom Smasher who reported the problem and provided an input file compressed in this way, which I did not previously have.

- Implement some improvements to file recovery logic.

2.0.3: February 15, 2009

- Compile cleanly with gcc 4.4.

- Handle strings encoded as UTF-16BE properly.

2.0.2: June 30, 2008

- Update test suite to work properly with a non-**bash** */bin/sh* and with Perl 5.10. No changes were made to the actual qpdf source code itself for this release.

2.0.1: May 6, 2008

- No changes in functionality or interface. This release includes fixes to the source code so that qpdf compiles properly and passes its test suite on a broader range of platforms. See *ChangeLog* in the source distribution for details.

2.0: April 29, 2008

- First public release.

Appendix B. Upgrading from 2.0 to 2.1

Although, as a general rule, we like to avoid introducing source-level incompatibilities in qpdf's interface, there were a few non-compatible changes made in this version. A considerable amount of source code that uses qpdf will probably compile without any changes, but in some cases, you may have to update your code. The changes are enumerated here. There are also some new interfaces; for those, please refer to the header files.

- QPDF's exception handling mechanism now uses **std::logic_error** for internal errors and **std::runtime_error** for runtime errors in favor of the now removed **QEXC** classes used in previous versions. The **QEXC** exception classes predated the addition of the *<stdexcept>* header file to the C++ standard library. Most of the exceptions thrown by the qpdf library itself are still of type **QPDFExc** which is now derived from **std::runtime_error**. Programs that caught an instance of **std::exception** and displayed it by calling the *what()* method will not need to be changed.

- The **QPDFExc** class now internally represents various fields of the error condition and provides interfaces for querying them. Among the fields is a numeric error code that can help applications act differently on (a small number of) different error conditions. See *QPDFExc.hh* for details.

- Warnings can be retrieved from qpdf as instances of **QPDFExc** instead of strings.

- The nested **QPDF::EncryptionData** class's constructor takes an additional argument. This class is primarily intended to be used by **QPDFWriter**. There's not really anything useful an end-user application could do with it. It probably shouldn't really be part of the public interface to begin with. Likewise, some of the methods for computing internal encryption dictionary parameters have changed to support /R=4 encryption.

- The method *QPDF::getUserPassword* has been removed since it didn't do what people would think it did. There are now two new methods: *QPDF::getPaddedUserPassword* and *QPDF::getTrimmedUserPassword*. The first one does what the old *QPDF::getUserPassword* method used to do, which is to return the password with possible binary padding as specified by the PDF specification. The second one returns a human-readable password string.

- The enumerated types that used to be nested in **QPDFWriter** have moved to top-level enumerated types and are now defined in the file *qpdf/Constants.h*. This enables them to be shared by both the C and C++ interfaces.

Appendix C. Upgrading to 3.0

For the most part, the API for qpdf version 3.0 is backward compatible with versions 2.1 and later. There are two exceptions:

- The method *QPDFObjectHandle::replaceStreamData* that uses a **StreamDataProvider** to provide the stream data no longer takes a *length* parameter. While it would have been easy enough to keep the parameter for backward compatibility, in this case, the parameter was removed since this provides the user an opportunity to simplify the calling code. This method was introduced in version 2.2. At the time, the *length* parameter was required in order to ensure that calls to the stream data provider returned the same length for a specific stream every time they were invoked. In particular, the linearization code depends on this. Instead, qpdf 3.0 and newer check for that constraint explicitly. The first time the stream data provider is called for a specific stream, the actual length is saved, and subsequent calls are required to return the same number of bytes. This means the calling code no longer has to compute the length in advance, which can be a significant simplification. If your code fails to compile because of the extra argument and you don't want to make other changes to your code, just omit the argument.

- Many methods take `long long` instead of other integer types. Most if not all existing code should compile fine with this change since such parameters had always previously been smaller types. This change was required to support files larger than two gigabytes in size.

Appendix D. Upgrading to 4.0

While version 4.0 includes a few non-compatible API changes, it is very unlikely that anyone's code would have used any of those parts of the API since they generally required information that would only be available inside the library. In the unlikely event that you should run into trouble, please see the ChangeLog. See also Appendix A, *Release Notes*, page 28 for a complete list of the non-compatible API changes made in this version.