QPDF Manual

For QPDF Version 2.0.3, February 15, 2009

Jay Berkenbilt

QPDF Manual: For QPDF Version 2.0.3, February 15, 2009

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.qbilt.org/.

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.

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, 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**.

Part of qpdf's test suite does comparisons of the contents PDF files by converting them images and comparing the images. You can optionally disable this part of the test suite by running **configure** with the **--disable-test-compare-images** flag. If you leave this enabled, 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://pages.cs.wisc.edu/~ghost/

This option is primarily intended for use by packagers of qpdf so that they can avoid having the qpdf packages depend on tiff and ghostscript software.

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 of 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.

There is currently no support for building qpdf on Windows. The code is reasonably portable, however, and making it work on Windows would probably be reasonably straightforward. A significant amount of the code in QPDF has been known to work on Windows in the past.

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.

outfilename does not have to be seekable, even when generating linearized files. Specifying "-" as **outfilename** means to write to standard output.

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.

--encrypt options --
Causes generation an encrypted output file. Please see Section 3.3, "Encryption Options," page 4 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.

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.

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 or 128. 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:

- **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

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

3.4. 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 9.

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 9.

3.5. 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.

-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 and linearization. A file for which **--check** reports no errors may still have errors in stream data but should otherwise be otherwise structurally sound.

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 unusuable 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.

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 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.

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.

There is no public interface for creating instances of QPDFObjectHandle. They can be created only inside the QPDF library. This is generally done through a call to the private method *QPDF::readObject* which uses ***QPDFTokenizer*** to read an indirect object at a given file position and return a ***QPDFObjectHandle*** that encapsulates it. There are also internal methods to create fabricated indirect objects from existing direct objects or to change an indirect object into a direct object, though these steps are not performed except to support rewriting.

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. 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.

6.4. 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 17 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 9.

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`.

6.5. 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 17. 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 21 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 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

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.