QPDF Manual

For QPDF Version 10.1.0, January 5, 2021

Jay Berkenbilt

QPDF Manual: For QPDF Version 10.1.0, January 5, 2021

Jay Berkenbilt

Copyright © 2005–2020 Jay Berkenbilt

Table of Contents

General Information v
1. What is QPDF? 1
2. Building and Installing QPDF 2
    2.1. System Requirements 2
    2.2. Build Instructions 2
    2.3. Crypto Providers 3
        2.3.1. Build Support For Crypto Providers 3
        2.3.2. Runtime Crypto Provider Selection 4
        2.3.3. Crypto Provider Information for Developers 4
        2.3.4. Crypto Provider Design Notes 4
    2.4. Notes for Packagers 5
3. Running QPDF 6
    3.1. Basic Invocation 6
        3.1.1. Exit Status 6
    3.2. Shell Completion 6
    3.3. Basic Options 7
    3.4. Encryption Options 11
    3.5. Page Selection Options 13
    3.6. Overlay and Underlay Options 15
    3.7. Advanced Parsing Options 16
    3.8. Advanced Transformation Options 17
    3.9. Testing, Inspection, and Debugging Options 22
    3.10. Unicode Passwords 24
4. QDF Mode 26
5. Using the QPDF Library 28
    5.1. Using QPDF from C++ 28
    5.2. Using QPDF from other languages 28
    5.3. A Note About Unicode File Names 28
6. QPDF JSON 30
    6.1. Overview 30
    6.2. JSON Guarantees 30
    6.3. Limitations of JSON Representation 31
    6.4. JSON: Special Considerations 31
7. Design and Library Notes 33
    7.1. Introduction 33
    7.2. Design Goals 33
    7.3. Helper Classes 34
    7.4. Implementation Notes 35
    7.5. Casting Policy 36
    7.6. Encryption 37
    7.7. Random Number Generation 38
    7.8. Adding and Removing Pages 38
    7.9. Reserving Object Numbers 38
    7.10. Copying Objects From Other PDF Files 39
    7.11. Writing PDF Files 39
    7.12. Filtered Streams 40
8. Linearization 41
    8.1. Basic Strategy for Linearization 41
    8.2. Preparing For Linearization 41
    8.3. Optimization 41
    8.4. Writing Linearized Files 42

8.5. Calculating Linearization Data 42
8.6. Known Issues with Linearization 42
8.7. Debugging Note 43
9. Object and Cross-Reference Streams 44
9.1. Object Streams 44
9.2. Cross-Reference Streams 44
9.2.1. Cross-Reference Stream Data 45
9.3. Implications for Linearized Files 45
9.4. Implementation Notes 46
A. Release Notes 47
B. Upgrading from 2.0 to 2.1 75
C. Upgrading to 3.0 76
D. Upgrading to 4.0 77

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 is licensed under the Apache License, Version 2.0 [http://www.apache.org/licenses/LICENSE-2.0] (the "License"). Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Versions of qpdf prior to version 7 were released under the terms of the Artistic License, version 2.0 [https://opensource.org/licenses/Artistic-2.0]. At your option, you may continue to consider qpdf to be licensed under those terms. The Apache License 2.0 permits everything that the Artistic License 2.0 permits but is slightly less restrictive. Allowing the Artistic License to continue being used is primary to help people who may have to get specific approval to use qpdf in their products.

QPDF is intentionally released with a permissive license. However, if there is some reason that the licensing terms don't work for your requirements, please feel free to contact the copyright holder to make other arrangements.

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.md* and *INSTALL* files in the source distribution.

2.1. System Requirements

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

- A C++ compiler that supports C++-11.
- zlib: http://www.zlib.net/
- jpeg: http://www.ijg.org/files/ or https://libjpeg-turbo.org/
- *Recommended but not required:* gnutls: https://www.gnutls.org/ to be able to use the gnutls crypto provider, and/or openssl: https://openssl.org/ to be able to use the openssl crypto provider.
- gnu make 3.81 or newer: http://www.gnu.org/software/make
- perl version 5.8 or newer: http://www.perl.org/; required for running the test suite. Starting with qpdf version 9.1.1, perl is no longer required at runtime.
- 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. 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.

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.md* and *INSTALL* files in the source distribution.

Building on Windows is a little bit more complicated. For details, please see *README-windows.md* 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.md*.

While `wchar_t` is part of the C++ standard, qpdf uses it in only one place in the public API, and it's just in a helper function. It is possible to build qpdf on a system that doesn't have `wchar_t`, and it's also possible to compile a program that uses qpdf on a system without `wchar_t` as long as you don't call that one method. This is a very unusual situation. For a detailed discussion, please see the top-level README.md file in qpdf's source distribution.

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*.

2.3. Crypto Providers

Starting with qpdf 9.1.0, the qpdf library can be built with multiple implementations of providers of cryptographic functions, which we refer to as “crypto providers.” At the time of writing, a crypto implementation must provide MD5 and SHA2 (256, 384, and 512-bit) hashes and RC4 and AES256 with and without CBC encryption. In the future, if digital signature is added to qpdf, there may be additional requirements beyond this.

Starting with qpdf version 9.1.0, the available implementations are `native` and `gnutls`. In qpdf 10.0.0, `openssl` was added. Additional implementations may be added if needed. It is also possible for a developer to provide their own implementation without modifying the qpdf library.

2.3.1. Build Support For Crypto Providers

When building with qpdf's build system, crypto providers can be enabled at build time using various **./configure** options. The default behavior is for **./configure** to discover which crypto providers can be supported based on available external libraries, to build all available crypto providers, and to use an external provider as the default over the native one. This behavior can be changed with the following flags to **./configure**:

- **--enable-crypto-*x*** (where *x* is a supported crypto provider): enable the *x* crypto provider, requiring any external dependencies it needs
- **--disable-crypto-*x***: disable the *x* provider, and do not link against its dependencies even if they are available
- **--with-default-crypto=*x***: make *x* the default provider even if a higher priority one is available
- **--disable-implicit-crypto**: only build crypto providers that are explicitly requested with an **--enable-crypto-*x*** option

For example, if you want to guarantee that the gnutls crypto provider is used and that the native provider is not built, you could run **./configure --enable-crypto-gnutls --disable-implicit-crypto**.

If you build qpdf using your own build system, in order for qpdf to work at all, you need to enable at least one crypto provider. The file *libqpdf/qpdf/qpdf-config.h.in* provides macros `DEFAULT_CRYPTO`, whose value must be a string

naming the default crypto provider, and various symbols starting with `USE_CRYPTO_`, at least one of which has to be enabled. Additionally, you must compile the source files that implement a crypto provider. To get a list of those files, look at *libqpdf/build.mk*. If you want to omit a particular crypto provider, as long as its `USE_CRYPTO_` symbol is undefined, you can completely ignore the source files that belong to a particular crypto provider. Additionally, crypto providers may have their own external dependencies that can be omitted if the crypto provider is not used. For example, if you are building qpdf yourself and are using an environment that does not support gnutls or openssl, you can ensure that `USE_CRYPTO_NATIVE` is defined, `USE_CRYPTO_GNUTLS` is not defined, and `DEFAULT_CRYPTO` is defined to `"native"`. Then you must include the source files used in the native implementation, some of which were added or renamed from earlier versions, to your build, and you can ignore *QPDFCrypto_gnutls.cc*. Always consult *libqpdf/build.mk* to get the list of source files you need to build.

2.3.2. Runtime Crypto Provider Selection

You can use the **--show-crypto** option to **qpdf** to get a list of available crypto providers. The default provider is always listed first, and the rest are listed in lexical order. Each crypto provider is listed on a line by itself with no other text, enabling the output of this command to be used easily in scripts.

You can override which crypto provider is used by setting the `QPDF_CRYPTO_PROVIDER` environment variable. There are few reasons to ever do this, but you might want to do it if you were explicitly trying to compare behavior of two different crypto providers while testing performance or reproducing a bug. It could also be useful for people who are implementing their own crypto providers.

2.3.3. Crypto Provider Information for Developers

If you are writing code that uses libqpdf and you want to force a certain crypto provider to be used, you can call the method *QPDFCryptoProvider::setDefaultProvider*. The argument is the name of a built-in or developer-supplied provider. To add your own crypto provider, you have to create a class derived from ***QPDFCryptoImpl*** and register it with ***QPDFCryptoProvider***. For additional information, see comments in *include/qpdf/QPDFCryptoImpl.hh*.

2.3.4. Crypto Provider Design Notes

This section describes a few bits of rationale for why the crypto provider interface was set up the way it was. You don't need to know any of this information, but it's provided for the record and in case it's interesting.

As a general rule, I want to avoid as much as possible including large blocks of code that are conditionally compiled such that, in most builds, some code is never built. This is dangerous because it makes it very easy for invalid code to creep in unnoticed. As such, I want it to be possible to build qpdf with all available crypto providers, and this is the way I build qpdf for local development. At the same time, if a particular packager feels that it is a security liability for qpdf to use crypto functionality from other than a library that gets considerable scrutiny for this specific purpose (such as gnutls, openssl, or nettle), then I want to give that packager the ability to completely disable qpdf's native implementation. Or if someone wants to avoid adding a dependency on one of the external crypto providers, I don't want the availability of the provider to impose additional external dependencies within that environment. Both of these are situations that I know to be true for some users of qpdf.

I want registration and selection of crypto providers to be thread-safe, and I want it to work deterministically for a developer to provide their own crypto provider and be able to set it up as the default. This was the primary motivation behind requiring C++-11 as doing so enabled me to exploit the guaranteed thread safety of local block static initialization. The ***QPDFCryptoProvider*** class uses a singleton pattern with thread-safe initialization to create the singleton instance of ***QPDFCryptoProvider*** and exposes only static methods in its public interface. In this way, if a developer wants to call any ***QPDFCryptoProvider*** methods, the library guarantees the ***QPDFCryptoProvider*** is fully initialized and all built-in crypto providers are registered. Making ***QPDFCryptoProvider*** actually know about all the built-in providers may seem a bit sad at first, but this choice makes it extremely clear exactly what the initialization behavior is. There's no question about provider implementations automatically registering themselves in a nondeter-

ministic order. It also means that implementations do not need to know anything about the provider interface, which makes them easier to test in isolation. Another advantage of this approach is that a developer who wants to develop their own crypto provider can do so in complete isolation from the qpdf library and, with just two calls, can make qpdf use their provider in their application. If they decided to contribute their code, plugging it into the qpdf library would require a very small change to qpdf's source code.

The decision to make the crypto provider selectable at runtime was one I struggled with a little, but I decided to do it for various reasons. Allowing an end user to switch crypto providers easily could be very useful for reproducing a potential bug. If a user reports a bug that some cryptographic thing is broken, I can easily ask that person to try with the `QPDF_CRYPTO_PROVIDER` variable set to different values. The same could apply in the event of a performance problem. This also makes it easier for qpdf's own test suite to exercise code with different providers without having to make every program that links with qpdf aware of the possibility of multiple providers. In qpdf's continuous integration environment, the entire test suite is run for each supported crypto provider. This is made simple by being able to select the provider using an environment variable.

Finally, making crypto providers selectable in this way establish a pattern that I may follow again in the future for stream filter providers. One could imagine a future enhancement where someone could provide their own implementations for basic filters like `/FlateDecode` or for other filters that qpdf doesn't support. Implementing the registration functions and internal storage of registered providers was also easier using C++-11's functional interfaces, which was another reason to require C++-11 at this time.

2.4. Notes for Packagers

If you are packaging qpdf for an operating system distribution, here are some things you may want to keep in mind:

- Starting in qpdf version 9.1.1, qpdf no longer has a runtime dependency on perl. This is because fix-qdf was rewritten in C++. However, qpdf still has a build-time dependency on perl.
- Make sure you are getting the intended behavior with regard to crypto providers. Read Section 2.3.1, “Build Support For Crypto Providers”, page 3 for details.
- Passing **--enable-show-failed-test-output** to **./configure** will cause any failed test output to be written to the console. This can be very useful for seeing test failures generated by autobuilders where you can't access qtest.log after the fact.
- If qpdf's build environment detects the presence of autoconf and related tools, it will check to ensure that automatically generated files are up-to-date with recorded checksums and fail if it detects a discrepancy. This feature is intended to prevent you from accidentally forgetting to regenerate automatic files after modifying their sources. If your packaging environment automatically refreshes automatic files, it can cause this check to fail. Suppress qpdf's checks by passing **--disable-check-autofiles** to **/.configure**. This is safe since qpdf's **autogen.sh** just runs autotools in the normal way.
- QPDF's **make install** does not install completion files by default, but as a packager, it's good if you install them wherever your distribution expects such files to go. You can find completion files to install in the *completions* directory.
- Packagers are encouraged to install the source files from the *examples* directory along with qpdf development packages.

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 3.5, “Page Selection Options”, page 13.

If **@filename** appears anywhere in the command-line, it will be read line by line, and each line will be treated as a command-line argument. The **@-** option allows arguments to be read from standard input. This allows qpdf to be invoked with an arbitrary number of arbitrarily long arguments. It is also very useful for avoiding having to pass passwords on the command line.

outfilename does not have to be seekable, even when generating linearized files. Specifying “**-**” as **outfilename** means to write to standard output. If you want to overwrite the input file with the output, use the option **--replace-input** and omit the output file name. You can't specify the same file as both the input and the output. If you do this, qpdf will tell you about the **--replace-input** option.

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

3.1.1. Exit Status

The exit status of **qpdf** may be interpreted as follows:

- `0`: no errors or warnings were found. The file may still have problems qpdf can't detect. If **--warning-exit-0** was specified, exit status 0 is used even if there are warnings.
- `2`: errors were found. qpdf was not able to fully process the file.
- `3`: qpdf encountered problems that it was able to recover from. In some cases, the resulting file may still be damaged. Note that qpdf still exits with status `3` if it finds warnings even when **--no-warn** is specified. With **--warning-exit-0**, warnings without errors exit with status 0 instead of 3.

Note that **qpdf** never exists with status `1`. If you get an exit status of `1`, it was something else, like the shell not being able to find or execute **qpdf**.

3.2. Shell Completion

Starting in qpdf version 8.3.0, qpdf provides its own completion support for zsh and bash. You can enable bash completion with **eval $(qpdf --completion-bash)** and zsh completion with **eval $(qpdf --completion-zsh)**. If **qpdf** is not in your path, you should invoke it above with an absolute path. If you invoke it with a relative path, it will warn you, and the completion won't work if you're in a different directory.

qpdf will use `argv[0]` to figure out where its executable is. This may produce unwanted results in some cases, especially if you are trying to use completion with copy of qpdf that is built from source. You can specify a full path to the qpdf you want to use for completion in the `QPDF_EXECUTABLE` environment variable.

3.3. Basic Options

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

--help

Display command-line invocation help.

--version

Display the current version of qpdf.

--copyright

Show detailed copyright information.

--show-crypto

Show a list of available crypto providers, each on a line by itself. The default provider is always listed first. See Section 2.3, “Crypto Providers”, page 3 for more information about crypto providers.

--completion-bash

Output a completion command you can eval to enable shell completion from bash.

--completion-zsh

Output a completion command you can eval to enable shell completion from zsh.

--password=password

Specifies a password for accessing encrypted files. Note that you can use **@filename** or **@-** as described above to put the password in a file or pass it via standard input so you can avoid specifying it on the command line.

--is-encrypted

Silently exit with status 0 if the file is encrypted or status 2 if the file is not encrypted. This is useful for shell scripts. Other options are ignored if this is given. This option is mutually exclusive with **--requires-password**. Both this option and **--requires-password** exit with status 2 for non-encrypted files.

--requires-password

Silently exit with status 0 if a password (other than as supplied) is required. Exit with status 2 if the file is not encrypted. Exit with status 3 if the file is encrypted but requires no password or the correct password has been supplied. This is useful for shell scripts. Note that any supplied password is used when opening the file. When used with a **--password** option, this option can be used to check the correctness of the password. In that case, an exit status of 3 means the file works with the supplied password. This option is mutually exclusive with **--is-encrypted**. Both this option and **--is-encrypted** exit with status 2 for non-encrypted files.

--verbose

Increase verbosity of output. For now, this just prints some indication of any file that it creates.

--progress

Indicate progress while writing files.

--no-warn

Suppress writing of warnings to stderr. If warnings were detected and suppressed, **qpdf** will still exit with exit code 3. See also **--warning-exit-0**.

--warning-exit-0

If warnings are found but no errors, exit with exit code 0 instead 3. When combined with **--no-warn**, the effect is for **qpdf** to completely ignore warnings.

--linearize

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

--replace-input

If specified, the output file name should be omitted. This option tells qpdf to replace the input file with the output. It does this by writing to *`infilename`.~qpdf-temp#* and, when done, overwriting the input file with the temporary file. If there were any warnings, the original input is saved as *`infilename`.~qpdf-orig*.

--copy-encryption=file

Encrypt the file using the same encryption parameters, including user and owner password, as the specified file. Use **--encryption-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.

--encryption-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 3.4, “Encryption Options”, page 11 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-is-hex-key

Overrides the usual computation/retrieval of the PDF file's encryption key from user/owner password with an explicit specification of the encryption key. When this option is specified, the argument to the **--password** option is interpreted as a hexadecimal-encoded key value. This only applies to the password used to open the main input file. It does not apply to other files opened by **--pages** or other options or to files being written.

Most users will never have a need for this option, and no standard viewers support this mode of operation, but it can be useful for forensic or investigatory purposes. For example, if a PDF file is encrypted with an unknown

password, a brute-force attack using the key directly is sometimes more efficient than one using the password. Also, if a file is heavily damaged, it may be possible to derive the encryption key and recover parts of the file using it directly. To expose the encryption key used by an encrypted file that you can open normally, use the **--show-encryption-key** option.

--suppress-password-recovery

Ordinarily, qpdf attempts to automatically compensate for passwords specified in the wrong character encoding. This option suppresses that behavior. Under normal conditions, there are no reasons to use this option. See Section 3.10, “Unicode Passwords”, page 24 for a discussion

--password-mode=*mode*

This option can be used to fine-tune how qpdf interprets Unicode (non-ASCII) password strings passed on the command line. With the exception of the **hex-bytes** mode, these only apply to passwords provided when encrypting files. The **hex-bytes** mode also applies to passwords specified for reading files. For additional discussion of the supported password modes and when you might want to use them, see Section 3.10, “Unicode Passwords”, page 24. The following modes are supported:

- **auto**: Automatically determine whether the specified password is a properly encoded Unicode (UTF-8) string, and transcode it as required by the PDF spec based on the type encryption being applied. On Windows starting with version 8.4.0, and on almost all other modern platforms, incoming passwords will be properly encoded in UTF-8, so this is almost always what you want.

- **unicode**: Tells qpdf that the incoming password is UTF-8, overriding whatever its automatic detection determines. The only difference between this mode and **auto** is that qpdf will fail with an error message if the password is not valid UTF-8 instead of falling back to **bytes** mode with a warning.

- **bytes**: Interpret the password as a literal byte string. For non-Windows platforms, this is what versions of qpdf prior to 8.4.0 did. For Windows platforms, there is no way to specify strings of binary data on the command line directly, but you can use the **@filename** option to do it, in which case this option forces qpdf to respect the string of bytes as provided. This option will allow you to encrypt PDF files with passwords that will not be usable by other readers.

- **hex-bytes**: Interpret the password as a hex-encoded string. This provides a way to pass binary data as a password on all platforms including Windows. As with **bytes**, this option may allow creation of files that can't be opened by other readers. This mode affects qpdf's interpretation of passwords specified for decrypting files as well as for encrypting them. It makes it possible to specify strings that are encoded in some manner other than the system's default encoding.

--rotate=[+|-]angle[:page-range]

Apply rotation to specified pages. The **page-range** portion of the option value has the same format as page ranges in Section 3.5, “Page Selection Options”, page 13. If the page range is omitted, the rotation is applied to all pages. The **angle** portion of the parameter may be either 90, 180, or 270. If preceded by **+** or **-**, the angle is added to or subtracted from the specified pages' original rotations. This is almost always what you want. Otherwise the pages' rotations are set to the exact value, which may cause the appearances of the pages to be inconsistent, especially for scans. For example, the command **qpdf in.pdf out.pdf --rotate=+90:2,4,6 --rotate=180:7-8** would rotate pages 2, 4, and 6 90 degrees clockwise from their original rotation and force the rotation of pages 7 through 9 to 180 degrees regardless of their original rotation, and the command **qpdf in.pdf out.pdf --rotate=+180** would rotate all pages by 180 degrees.

--keep-files-open=*[yn]*

This option controls whether qpdf keeps individual files open while merging. Prior to version 8.1.0, qpdf always kept all files open, but this meant that the number of files that could be merged was limited by the operating

system's open file limit. Version 8.1.0 opened files as they were referenced and closed them after each read, but this caused a major performance impact. Version 8.2.0 optimized the performance but did so in a way that, for local file systems, there was a small but unavoidable performance hit, but for networked file systems, the performance impact could be very high. Starting with version 8.2.1, the default behavior is that files are kept open if no more than 200 files are specified, but that the behavior can be explicitly overridden with the **--keep-files-open** flag. If you are merging more than 200 files but less than the operating system's max open files limit, you may want to use **--keep-files-open=y**, especially if working over a networked file system. If you are using a local file system where the overhead is low and you might sometimes merge more than the OS limit's number of files from a script and are not worried about a few seconds additional processing time, you may want to specify **--keep-files-open=n**. The threshold for switching may be changed from the default 200 with the **--keep-files-open-threshold** option.

--keep-files-open-threshold=*count*

If specified, overrides the default value of 200 used as the threshold for qpdf deciding whether or not to keep files open. See **--keep-files-open** for details.

--pages options --

Select specific pages from one or more input files. See Section 3.5, “Page Selection Options”, page 13 for details on how to do page selection (splitting and merging).

--collate

When specified, collate rather than concatenate pages from files specified with **--pages**. See Section 3.5, “Page Selection Options”, page 13 for additional details.

--flatten-rotation

For each page that is rotated using the `/Rotate` key in the page's dictionary, remove the `/Rotate` key and implement the identical rotation semantics by modifying the page's contents. This option can be useful to prepare files for buggy PDF applications that don't properly handle rotated pages.

--split-pages=[n]

Write each group of **n** pages to a separate output file. If **n** is not specified, create single pages. Output file names are generated as follows:

- If the string `%d` appears in the output file name, it is replaced with a range of zero-padded page numbers starting from 1.
- Otherwise, if the output file name ends in *.pdf* (case insensitive), a zero-padded page range, preceded by a dash, is inserted before the file extension.
- Otherwise, the file name is appended with a zero-padded page range preceded by a dash.

Page ranges are a single number in the case of single-page groups or two numbers separated by a dash otherwise. For example, if *infile.pdf* has 12 pages

- **qpdf --split-pages infile.pdf %d-out** would generate files *01-out* through *12-out*
- **qpdf --split-pages=2 infile.pdf outfile.pdf** would generate files *outfile-01-02.pdf* through *outfile-11-12.pdf*
- **qpdf --split-pages infile.pdf something.else** would generate files *something.else-01* through *something.else-12*

Note that outlines, threads, and other global features of the original PDF file are not preserved. For each page of output, this option creates an empty PDF and copies a single page from the output into it. If you require the

global data, you will have to run **qpdf** with the **--pages** option once for each file. Using **--split-pages** is much faster if you don't require the global data.

--overlay options --

Overlay pages from another file onto the output pages. See Section 3.6, “Overlay and Underlay Options”, page 15 for details on overlay/underlay.

--underlay options --

Overlay pages from another file onto the output pages. See Section 3.6, “Overlay and Underlay Options”, page 15 for details on overlay/underlay.

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.

Prior to 8.4.0, in the case of passwords that contain characters that fall outside of 7-bit US-ASCII, qpdf left the burden of supplying properly encoded encryption and decryption passwords to the user. Starting in qpdf 8.4.0, qpdf does this automatically in most cases. For an in-depth discussion, please see Section 3.10, “Unicode Passwords”, page 24. Previous versions of this manual described workarounds using the **iconv** command. Such workarounds are no longer required or recommended with qpdf 8.4.0. However, for backward compatibility, qpdf attempts to detect those workarounds and do the right thing in most cases.

3.4. 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. The qpdf library disregards this field when AES is used or when 256-bit encryption is used. You should really never disable accessibility, but qpdf lets you do it in case you need to configure a file this way for testing purposes. The PDF spec says that conforming readers should disregard this permission and always allow accessibility.

--extract=[yn]

Determines whether or not to allow text/graphic extraction.

--assemble=[yn]

Determines whether document assembly (rotation and reordering of pages) is allowed.

--annotate=[yn]

Determines whether modifying annotations is allowed. This includes adding comments and filling in form fields. Also allows editing of form fields if **--modify-other=y** is given.

--form=[yn]

Determines whether filling form fields is allowed.

--modify-other=[yn]

Allow all document editing except those controlled separately by the **--assemble**, **--annotate**, and **--form** options.

--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. This way of controlling modify access has less granularity than new options added in qpdf 8.4. ***modify-opt*** may be one of the following:

- **all**: allow full document modification
- **annotate**: allow comment authoring, form operations, and document assembly
- **form**: allow form field fill-in and signing and document assembly
- **assembly**: allow document assembly only
- **none**: allow no modifications

Using the **--modify** option does not allow you to create certain combinations of permissions such as allowing form filling but not allowing document assembly. Starting with qpdf 8.4, you can either just use the other options to control fields individually, or you can use something like **--modify=form --assembly=n** to fine tune.

--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.

3.5. 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.

Starting with qpf 8.4, the special input file name "." can be used shortcut for the primary input filename.

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.

Starting with qpdf 5.0.0, it is possible to omit the page range. If qpdf sees a value in the place where it expects a page range and that value is not a valid range but is a valid file name, qpdf will implicitly use the range `1-z`, meaning that it will include all pages in the file. This makes it possible to easily combine all pages in a set of files with a command like **qpdf --empty out.pdf --pages *.pdf --**.

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. A number preceded by an "r" indicates to count from the end, so `r3-r1` would be the last three pages of the document. 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. Numbers may be repeated in a page range. A page range may be optionally appended with `:even` or `:odd` to indicate only the even or odd pages in the given range. Note that even and odd refer to the positions within the specified, range, not whether the original number is even or odd.

Example page ranges:

- `1,3,5-9,15-12`: pages 1, 3, 5, 6, 7, 8, 9, 15, 14, 13, and 12 in that order.
- `z-1`: all pages in the document in reverse
- `r3-r1`: the last three pages of the document
- `r1-r3`: the last three pages of the document in reverse order
- `1-20:even`: even pages from 2 to 20
- `5,7-9,12:odd`: pages 5, 8, and, 12, which are the pages in odd positions from among the original range, which represents pages 5, 7, 8, 9, and 12.

Starting in qpdf version 8.3, you can specify the **--collate** option. Note that this option is specified outside of **--pages ... --**. When **--collate** is specified, it changes the meaning of **--pages** so that the specified files, as modified by page ranges, are collated rather than concatenated. For example, if you add the files *odd.pdf* and *even.pdf* containing odd and even pages of a document respectively, you could run **qpdf --collate odd.pdf --pages odd.pdf even.pdf --all.pdf** to collate the pages. This would pick page 1 from odd, page 1 from even, page 2 from odd, page 2 from even, etc. until all pages have been included. Any number of files and page ranges can be specified. If any file has fewer pages, that file is just skipped when its pages have all been included. For example, if you ran **qpdf --collate --empty --pages a.pdf 1-5 b.pdf 6-4 c.pdf r1 -- out.pdf**, you would get the following pages in this order:

- a.pdf page 1
- b.pdf page 6
- c.pdf last page
- a.pdf page 2
- b.pdf page 5
- a.pdf page 3
- b.pdf page 4
- a.pdf page 4

- a.pdf page 5

Starting in qpdf version 8.3, when you split and merge files, any page labels (page numbers) are preserved in the final file. It is expected that more document features will be preserved by splitting and merging. In the mean time, 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. 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 . 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 . 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 document-level metadata 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.

Prior to version 8.4, it was not possible to specify the same page from the same file directly more than once, and the workaround of specifying the same file in more than one way was required. Version 8.4 removes this limitation, but there is still a valid use case. When you specify the same page from the same file more than once, qpdf will share objects between the pages. If you are going to do further manipulation on the file and need the two instances of the same original page to be deep copies, then you can specify the file in two different ways. For example **qpdf in.pdf --pages . 1 ./in.pdf 1 -- out.pdf** would create a file with two copies of the first page of the input, and the two copies would share any objects in common. This includes fonts, images, and anything else the page references.

3.6. Overlay and Underlay Options

Starting with qpdf 8.4, it is possible to overlay or underlay pages from other files onto the output generated by qpdf. Specify overlay or underlay as follows:

```
{ --overlay | --underlay } file [ options ] --
```

Overlay and underlay options are processed late, so they can be combined with other like merging and will apply to the final output. The **--overlay** and **--underlay** options work the same way, except underlay pages are drawn underneath

the page to which they are applied, possibly obscured by the original page, and overlay files are drawn on top of the page to which they are applied, possibly obscuring the page. You can combine overlay and underlay.

The default behavior of overlay and underlay is that pages are taken from the overlay/underlay file in sequence and applied to corresponding pages in the output until there are no more output pages. If the overlay or underlay file runs out of pages, remaining output pages are left alone. This behavior can be modified by options, which are provided between the **--overlay** or **--underlay** flag and the **--** option. The following options are supported:

- **--password=password**: supply a password if the overlay/underlay file is encrypted.
- **--to=page-range**: a range of pages in the same form at described in Section 3.5, “Page Selection Options”, page 13 indicates which pages in the output should have the overlay/underlay applied. If not specified, overlay/underlay are applied to all pages.
- **--from=[page-range]**: a range of pages that specifies which pages in the overlay/underlay file will be used for overlay or underlay. If not specified, all pages will be used. This can be explicitly specified to be empty if **--repeat** is used.
- **--repeat=page-range**: an optional range of pages that specifies which pages in the overlay/underlay file will be repeated after the “from” pages are used up. If you want to repeat a range of pages starting at the beginning, you can explicitly use **--from=**.

Here are some examples.

- **--overlay o.pdf --to=1-5 --from=1-3 --repeat=4 --**: overlay the first three pages from file *o.pdf* onto the first three pages of the output, then overlay page 4 from *o.pdf* onto pages 4 and 5 of the output. Leave remaining output pages untouched.
- **--underlay footer.pdf --from= --repeat=1,2 --**: Underlay page 1 of *footer.pdf* on all odd output pages, and underlay page 2 of *footer.pdf* on all even output pages.

3.7. Advanced Parsing Options

These options control aspects of how qpdf reads PDF files. Mostly these are of use to people who are working with damaged files. There is little reason to use these options unless you are trying to solve specific problems. The following options are available:

--suppress-recovery

Prevents qpdf from attempting to recover damaged files.

--ignore-xref-streams

Tells qpdf to ignore any cross-reference streams.

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.

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.

3.8. 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:

--compress-streams=*[yn]*

By default, or with **--compress-streams=y**, qpdf will compress any stream with no other filters applied to it with the `/FlateDecode` filter when it writes it. To suppress this behavior and preserve uncompressed streams as uncompressed, use **--compress-streams=n**.

--decode-level=*option*

Controls which streams qpdf tries to decode. The default is **generalized**. The following options are available:

- **none**: do not attempt to decode any streams
- **generalized**: decode streams filtered with supported generalized filters: `/LZWDecode`, `/FlateDecode`, `/ASCII85Decode`, and `/ASCIIHexDecode`. We define generalized filters as those to be used for general-purpose compression or encoding, as opposed to filters specifically designed for image data. Note that, by default, streams already compressed with `/FlateDecode` are not uncompressed and recompressed unless you also specify **--recompress-flate**.
- **specialized**: in addition to generalized, decode streams with supported non-lossy specialized filters; currently this is just `/RunLengthDecode`
- **all**: in addition to generalized and specialized, decode streams with supported lossy filters; currently this is just `/DCTDecode` (JPEG)

--stream-data=*option*

Controls transformation of stream data. This option predates the **--compress-streams** and **--decode-level** options. Those options can be used to achieve the same affect with more control. The value of ***option*** may be one of the following:

- **compress**: recompress stream data when possible (default); equivalent to **--compress-streams=y --decode-level=generalized**. Does not recompress streams already compressed with `/FlateDecode` unless **--recompress-flate** is also specified.
- **preserve**: leave all stream data as is; equivalent to **--compress-streams=n --decode-level=none**
- **uncompress**: uncompress stream data compressed with generalized filters when possible; equivalent to **--compress-streams=n --decode-level=generalized**

--recompress-flate

By default, streams already compressed with `/FlateDecode` are left alone rather than being uncompressed and recompressed. This option causes qpdf to uncompress and recompress the streams. There is a significant performance cost to using this option, but you probably want to use it if you specify **--compression-level**.

--compression-level=*level*

When writing new streams that are compressed with `/FlateDecode`, use the specified compression level. The value of **level** should be a number from 1 to 9 and is passed directly to zlib, which implements deflate compression. Note that qpdf doesn't uncompress and recompress streams by default. To have this option apply to already compressed streams, you should also specify **--recompress-flate**. If your goal is to shrink the size of PDF files, you should also use **--object-streams=generate**.

--normalize-content=[yn]

Enables or disables normalization of content streams. Content normalization is enabled by default in QDF mode. Please see Chapter 4, *QDF Mode*, page 26 for additional discussion of QDF mode.

--object-streams=*mode*

Controls handling 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

--preserve-unreferenced

Tells qpdf to preserve objects that are not referenced when writing the file. Ordinarily any object that is not referenced in a traversal of the document from the trailer dictionary will be discarded. This may be useful in working with some damaged files or inspecting files with known unreferenced objects.

This flag is ignored for linearized files and has the effect of causing objects in the new file to be written in order by object ID from the original file. This does not mean that object numbers will be the same since qpdf may create stream lengths as direct or indirect differently from the original file, and the original file may have gaps in its numbering.

See also **--preserve-unreferenced-resources**, which does something completely different.

--remove-unreferenced-resources=*option*

The *option* may be `auto`, `yes`, or `no`. The default is `auto`.

Starting with qpdf 8.1, when splitting pages, qpdf is able to attempt to remove images and fonts that are not used by a page even if they are referenced in the page's resources dictionary. When shared resources are in use, this behavior can greatly reduce the file sizes of split pages, but the analysis is very slow. In versions from 8.1 through 9.1.1, qpdf did this analysis by default. Starting in qpdf 10.0.0, if `auto` is used, qpdf does a quick analysis of the file to determine whether the file is likely to have unreferenced objects on pages, a pattern that frequently occurs when resource dictionaries are shared across multiple pages and rarely occurs otherwise. If it discovers this pattern, then it will attempt to remove unreferenced resources. Usually this means you get the slower splitting speed only when it's actually going to create smaller files. You can suppress removal of unreferenced resources altogether by specifying `no` or force it to do the full algorithm by specifying `yes`.

Other than cases in which you don't care about file size and care a lot about runtime, there are few reasons to use this option, especially now that `auto` mode is supported. One reason to use this is if you suspect that qpdf is removing resources it shouldn't be removing. If you encounter that case, please report it as bug at https://github.com/qpdf/qpdf/issues/.

--preserve-unreferenced-resources

This is a synonym for **--remove-unreferenced-resources=no**.

See also **--preserve-unreferenced**, which does something completely different.

--newline-before-endstream

Tells qpdf to insert a newline before the `endstream` keyword, not counted in the length, after any stream content even if the last character of the stream was a newline. This may result in two newlines in some cases. This is

a requirement of PDF/A. While qpdf doesn't specifically know how to generate PDF/A-compliant PDFs, this at least prevents it from removing compliance on already compliant files.

--linearize-pass1=*file*

Write the first pass of linearization to the named file. The resulting file is not a valid PDF file. This option is useful only for debugging ***QPDFWriter***'s linearization code. When qpdf linearizes files, it writes the file in two passes, using the first pass to calculate sizes and offsets that are required for hint tables and the linearization dictionary. Ordinarily, the first pass is discarded. This option enables it to be captured.

--coalesce-contents

When a page's contents are split across multiple streams, this option causes qpdf to combine them into a single stream. Use of this option is never necessary for ordinary usage, but it can help when working with some files in some cases. For example, this can also be combined with QDF mode or content normalization to make it easier to look at all of a page's contents at once.

--flatten-annotations=*option*

This option collapses annotations into the pages' contents with special handling for form fields. Ordinarily, an annotation is rendered separately and on top of the page. Combining annotations into the page's contents effectively freezes the placement of the annotations, making them look right after various page transformations. The library functionality backing this option was added for the benefit of programs that want to create *n-up* page layouts and other similar things that don't work well with annotations. The *option* parameter may be any of the following:

- **all**: include all annotations that are not marked invisible or hidden
- **print**: only include annotations that indicate that they should appear when the page is printed
- **screen**: omit annotations that indicate they should not appear on the screen

Note that form fields are special because the annotations that are used to render filled-in form fields may become out of date from the fields' values if the form is filled in by a program that doesn't know how to update the appearances. If qpdf detects this case, its default behavior is not to flatten those annotations because doing so would cause the value of the form field to be lost. This gives you a chance to go back and resave the form with a program that knows how to generate appearances. QPDF itself can generate appearances with some limitations. See the **--generate-appearances** option below.

--generate-appearances

If a file contains interactive form fields and indicates that the appearances are out of date with the values of the form, this flag will regenerate appearances, subject to a few limitations. Note that there is not usually a reason to do this, but it can be necessary before using the **--flatten-annotations** option. Most of these are not a problem with well-behaved PDF files. The limitations are as follows:

- Radio button and checkbox appearances use the pre-set values in the PDF file. QPDF just makes sure that the correct appearance is displayed based on the value of the field. This is fine for PDF files that create their forms properly. Some PDF writers save appearances for fields when they change, which could cause some controls to have inconsistent appearances.
- For text fields and list boxes, any characters that fall outside of US-ASCII or, if detected, “Windows ANSI” or “Mac Roman” encoding, will be replaced by the ? character.
- Quadding is ignored. Quadding is used to specify whether the contents of a field should be left, center, or right aligned with the field.
- Rich text, multi-line, and other more elaborate formatting directives are ignored.

- There is no support for multi-select fields or signature fields.

If qpdf doesn't do a good enough job with your form, use an external application to save your filled-in form before processing it with qpdf.

--optimize-images

This flag causes qpdf to recompress all images that are not compressed with DCT (JPEG) using DCT compression as long as doing so decreases the size in bytes of the image data and the image does not fall below minimum specified dimensions. Useful information is provided when used in combination with **--verbose**. See also the **--oi-min-width**, **--oi-min-height**, and **--oi-min-area** options. By default, starting in qpdf 8.4, inline images are converted to regular images and optimized as well. Use **--keep-inline-images** to prevent inline images from being included.

--oi-min-width=*width*

Avoid optimizing images whose width is below the specified amount. If omitted, the default is 128 pixels. Use 0 for no minimum.

--oi-min-height=*height*

Avoid optimizing images whose height is below the specified amount. If omitted, the default is 128 pixels. Use 0 for no minimum.

--oi-min-area=*area-in-pixels*

Avoid optimizing images whose pixel count (width × height) is below the specified amount. If omitted, the default is 16,384 pixels. Use 0 for no minimum.

--externalize-inline-images

Convert inline images to regular images. By default, images whose data is at least 1,024 bytes are converted when this option is selected. Use **--ii-min-bytes** to change the size threshold. This option is implicitly selected when **--optimize-images** is selected. Use **--keep-inline-images** to exclude inline images from image optimization.

--ii-min-bytes=*bytes*

Avoid converting inline images whose size is below the specified minimum size to regular images. If omitted, the default is 1,024 bytes. Use 0 for no minimum.

--keep-inline-images

Prevent inline images from being included in image optimization. This option has no affect when **--optimize-images** is not specified.

--remove-page-labels

Remove page labels from the output file.

--qdf

Turns on QDF mode. For additional information on QDF, please see Chapter 4, *QDF Mode*, page 26. Note that **--linearize** disables QDF mode.

--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 **--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.

When normalizing content, if qpdf runs into any lexical errors, it will print a warning indicating that content may be damaged. The only situation in which qpdf is known to cause damage during content normalization is when a page's contents are split across multiple streams and streams are split in the middle of a lexical token such as a string, name, or inline image. Note that files that do this are invalid since the PDF specification states that content streams are not to be split in the middle of a token. If you want to inspect the original content streams in an uncompressed format, you can always run with **--qdf --normalize-content=n** for a QDF file without content normalization, or alternatively **--stream-data=uncompress** for a regular non-QDF mode file with uncompressed streams. These will both uncompress all the streams but will not attempt to normalize content. Please note that if you are using content normalization or QDF mode for the purpose of manually inspecting files, you don't have to care about this.

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.

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 26.

3.9. 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:

--deterministic-id

Causes generation of a deterministic value for /ID. This prevents use of timestamp and output file name information in the /ID generation. Instead, at some slight additional runtime cost, the /ID field is generated to include a digest of the significant parts of the content of the output PDF file. This means that a given qpdf operation should generate the same /ID each time it is run, which can be useful when caching results or for generation of some test data. Use of this flag is not compatible with creation of encrypted files.

--static-id

Causes generation of a fixed value for /ID. This is intended for testing only. Never use it for production files. If you are trying to get the same /ID each time for a given file and you are not generating encrypted files, consider using the **--deterministic-id** option.

--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.

--show-encryption-key

When encryption information is being displayed, as when **--check** or **--show-encryption** is given, display the computed or retrieved encryption key as a hexadecimal string. This value is not ordinarily useful to users, but it can be used as the argument to **--password** if the **--password-is-hex-key** is specified. Note that, when PDF files are encrypted, passwords and other metadata are used only to compute an encryption key, and the encryption key is what is actually used for encryption. This enables retrieval of that key.

--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=trailer|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-npages

Prints the number of pages in the input file on a line by itself. Since the number of pages appears by itself on a line, this option can be useful for scripting if you need to know the number of pages in a file.

--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.)

--json

Generate a JSON representation of the file. This is described in depth in Chapter 6, *QPDF JSON*, page 30

--json-help

Describe the format of the JSON output.

--json-key=key

This option is repeatable. If specified, only top-level keys specified will be included in the JSON output. If not specified, all keys will be shown.

--json-object=trailer|obj[,gen]

This option is repeatable. If specified, only specified objects will be shown in the “`objects`” key of the JSON output. If absent, all objects will be shown.

--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). When **--check** is combined with other options, checks are always performed before any other options are processed. For erroneous files, **--check** will cause qpdf to attempt to recover, after which other options are effectively operating on the recovered file. Combining **--check** with other options in this way can be useful for manually recovering severely damaged files.

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.

3.10. Unicode Passwords

At the library API level, all methods that perform encryption and decryption interpret passwords as strings of bytes. It is up to the caller to ensure that they are appropriately encoded. Starting with qpdf version 8.4.0, qpdf will attempt to make this easier for you when interact with qpdf via its command line interface. The PDF specification requires passwords used to encrypt files with 40-bit or 128-bit encryption to be encoded with PDF Doc encoding. This encoding is a single-byte encoding that supports ISO-Latin-1 and a handful of other commonly used characters. It has a large overlap with Windows ANSI but is not exactly the same. There is generally not a way to provide PDF Doc encoded strings on the command line. As such, qpdf versions prior to 8.4.0 would often create PDF files that couldn't be opened with other software when given a password with non-ASCII characters to encrypt a file with 40-bit or 128-bit encryption. Starting with qpdf 8.4.0, qpdf recognizes the encoding of the parameter and transcodes it as needed. The rest of this section provides the details about exactly how qpdf behaves. Most users will not need to know this information, but it might be useful if you have been working around qpdf's old behavior or if you are using qpdf to generate encrypted files for testing other PDF software.

A note about Windows: when qpdf builds, it attempts to determine what it has to do to use *wmain* instead of *main* on Windows. The *wmain* function is an alternative entry point that receives all arguments as UTF-16-encoded strings. When qpdf starts up this way, it converts all the strings to UTF-8 encoding and then invokes the regular main. This means that, as far as qpdf is concerned, it receives its command-line arguments with UTF-8 encoding, just as it would in any modern Linux or UNIX environment.

If a file is being encrypted with 40-bit or 128-bit encryption and the supplied password is not a valid UTF-8 string, qpdf will fall back to the behavior of interpreting the password as a string of bytes. If you have old scripts that encrypt files by passing the output of **iconv** to qpdf, you no longer need to do that, but if you do, qpdf should still work. The only exception would be for the extremely unlikely case of a password that is encoded with a single-byte encoding but also happens to be valid UTF-8. Such a password would contain strings of even numbers of characters that alternate between accented letters and symbols. In the extremely unlikely event that you are intentionally using such passwords and qpdf is thwarting you by interpreting them as UTF-8, you can use **--password-mode=bytes** to suppress qpdf's automatic behavior.

The **--password-mode** option, as described earlier in this chapter, can be used to change qpdf's interpretation of supplied passwords. There are very few reasons to use this option. One would be the unlikely case described in the previous paragraph in which the supplied password happens to be valid UTF-8 but isn't supposed to be UTF-8. Your best bet would be just to provide the password as a valid UTF-8 string, but you could also use **--password-mode=bytes**. Another reason to use **--password-mode=bytes** would be to intentionally generate PDF files encrypted with pass-

words that are not properly encoded. The qpdf test suite does this to generate invalid files for the purpose of testing its password recovery capability. If you were trying to create intentionally incorrect files for a similar purposes, the **bytes** password mode can enable you to do this.

When qpdf attempts to decrypt a file with a password that contains non-ASCII characters, it will generate a list of alternative passwords by attempting to interpret the password as each of a handful of different coding systems and then transcode them to the required format. This helps to compensate for the supplied password being given in the wrong coding system, such as would happen if you used the **iconv** workaround that was previously needed. It also generates passwords by doing the reverse operation: translating from correct in incorrect encoding of the password. This would enable qpdf to decrypt files using passwords that were improperly encoded by whatever software encrypted the files, including older versions of qpdf invoked without properly encoded passwords. The combination of these two recovery methods should make qpdf transparently open most encrypted files with the password supplied correctly but in the wrong coding system. There are no real downsides to this behavior, but if you don't want qpdf to do this, you can use the **--suppress-password-recovery** option. One reason to do that is to ensure that you know the exact password that was used to encrypt the file.

With these changes, qpdf now generates compliant passwords in most cases. There are still some exceptions. In particular, the PDF specification directs compliant writers to normalize Unicode passwords and to perform certain transformations on passwords with bidirectional text. Implementing this functionality requires using a real Unicode library like ICU. If a client application that uses qpdf wants to do this, the qpdf library will accept the resulting passwords, but qpdf will not perform these transformations itself. It is possible that this will be addressed in a future version of qpdf. The ***QPDFWriter*** methods that enable encryption on the output file accept passwords as strings of bytes.

Please note that the **--password-is-hex-key** option is unrelated to all this. This flag bypasses the normal process of going from password to encryption string entirely, allowing the raw encryption key to be specified directly. This is useful for forensic purposes or for brute-force recovery of files with unknown passwords.

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). Note that QDF mode does not support linearized files. If you enable linearization, QDF mode is automatically disabled.

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

5.1. Using QPDF from C++

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/QPDFWriter.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 `-lz -ljpeg` 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.

5.2. Using QPDF from other languages

The qpdf library is implemented in C++, which makes it hard to use directly in other languages. There are a few things that can help.

"C"

> The qpdf library includes a "C" language interface that provides a subset of the overall capabilities. The header file *qpdf/qpdf-c.h* includes information about its use. As long as you use a C++ linker, you can link C programs with qpdf and use the C API. For languages that can directly load methods from a shared library, the C API can also be useful. People have reported success using the C API from other languages on Windows by directly calling functions in the DLL.

Python

> A Python module called pikepdf [https://pypi.org/project/pikepdf/] provides a clean and highly functional set of Python bindings to the qpdf library. Using pikepdf, you can work with PDF files in a natural way and combine qpdf's capabilities with other functionality provided by Python's rich standard library and available modules.

Other Languages

> Starting with version 8.3.0, the **qpdf** command-line tool can produce a JSON representation of the PDF file's non-content data. This can facilitate interacting programmatically with PDF files through qpdf's command line interface. For more information, please see Chapter 6, *QPDF JSON*, page 30.

5.3. A Note About Unicode File Names

When strings are passed to qpdf library routines either as `char*` or as `std::string`, they are treated as byte arrays except where otherwise noted. When Unicode is desired, qpdf wants UTF-8 unless otherwise noted in comments in header files. In modern UNIX/Linux environments, this generally does the right thing. In Windows, it's a bit more complicated. Starting in qpdf 8.4.0, passwords that contain Unicode characters are handled much better, and starting in qpdf 8.4.1, the library attempts to properly handle Unicode characters in filenames. In particular, in Windows, if a UTF-8 encoded string is used as a filename in either ***QPDF*** or ***QPDFWriter***, it is internally converted to `wchar_t*`,

and Unicode-aware Windows APIs are used. As such, qpdf will generally operate properly on files with non-ASCII characters in their names as long as the filenames are UTF-8 encoded for passing into the qpdf library API, but there are still some rough edges, such as the encoding of the filenames in error messages our CLI output messages. Patches or bug reports are welcome for any continuing issues with Unicode file names in Windows.

Chapter 6. QPDF JSON

6.1. Overview

Beginning with qpdf version 8.3.0, the **qpdf** command-line program can produce a JSON representation of the non-content data in a PDF file. It includes a dump in JSON format of all objects in the PDF file excluding the content of streams. This JSON representation makes it very easy to look in detail at the structure of a given PDF file, and it also provides a great way to work with PDF files programmatically from the command-line in languages that can't call or link with the qpdf library directly. Note that stream data can be extracted from PDF files using other qpdf command-line options.

6.2. JSON Guarantees

The qpdf JSON representation includes a JSON serialization of the raw objects in the PDF file as well as some computed information in a more easily extracted format. QPDF provides some guarantees about its JSON format. These guarantees are designed to simplify the experience of a developer working with the JSON format.

Compatibility

The top-level JSON object output is a dictionary. The JSON output contains various nested dictionaries and arrays. With the exception of dictionaries that are populated by the fields of objects from the file, all instances of a dictionary are guaranteed to have exactly the same keys. Future versions of qpdf are free to add additional keys but not to remove keys or change the type of object that a key points to. The qpdf program validates this guarantee, and in the unlikely event that a bug in qpdf should cause it to generate data that doesn't conform to this rule, it will ask you to file a bug report.

The top-level JSON structure contains a "`version`" key whose value is simple integer. The value of the `version` key will be incremented if a non-compatible change is made. A non-compatible change would be any change that involves removal of a key, a change to the format of data pointed to by a key, or a semantic change that requires a different interpretation of a previously existing key. A strong effort will be made to avoid breaking compatibility.

Documentation

The **qpdf** command can be invoked with the **--json-help** option. This will output a JSON structure that has the same structure as the JSON output that qpdf generates, except that each field in the help output is a description of the corresponding field in the JSON output. The specific guarantees are as follows:

- A dictionary in the help output means that the corresponding location in the actual JSON output is also a dictionary with exactly the same keys; that is, no keys present in help are absent in the real output, and no keys will be present in the real output that are not in help. As a special case, if the dictionary has a single key whose name starts with `<` and ends with `>`, it means that the JSON output is a dictionary that can have any keys, each of which conforms to the value of the special key. This is used for cases in which the keys of the dictionary are things like object IDs.

- A string in the help output is a description of the item that appears in the corresponding location of the actual output. The corresponding output can have any format.

- An array in the help output always contains a single element. It indicates that the corresponding location in the actual output is also an array, and that each element of the array has whatever format is implied by the single element of the help output's array.

For example, the help output indicates includes a "`pagelabels`" key whose value is an array of one element. That element is a dictionary with keys "`index`" and "`label`". In addition to describing the meaning of those keys, this tells you that the actual JSON output will contain a `pagelabels` array, each of whose elements is a dictionary that contains an `index` key, a `label` key, and no other keys.

Directness and Simplicity

The JSON output contains the value of every object in the file, but it also contains some processed data. This is analogous to how qpdf's library interface works. The processed data is similar to the helper functions in that it allows you to look at certain aspects of the PDF file without having to understand all the nuances of the PDF specification, while the raw objects allow you to mine the PDF for anything that the higher-level interfaces are lacking.

6.3. Limitations of JSON Representation

There are a few limitations to be aware of with the JSON structure:

- Strings, names, and indirect object references in the original PDF file are all converted to strings in the JSON representation. In the case of a "normal" PDF file, you can tell the difference because a name starts with a slash (`/`), and an indirect object reference looks like `n n R`, but if there were to be a string that looked like a name or indirect object reference, there would be no way to tell this from the JSON output. Note that there are certain cases where you know for sure what something is, such as knowing that dictionary keys in objects are always names and that certain things in the higher-level computed data are known to contain indirect object references.

- The JSON format doesn't support binary data very well. Mostly the details are not important, but they are presented here for information. When qpdf outputs a string in the JSON representation, it converts the string to UTF-8, assuming usual PDF string semantics. Specifically, if the original string is UTF-16, it is converted to UTF-8. Otherwise, it is assumed to have PDF doc encoding, and is converted to UTF-8 with that assumption. This causes strange things to happen to binary strings. For example, if you had the binary string `<038051>`, this would be output to the JSON as `\u0003•Q` because `03` is not a printable character and `80` is the bullet character in PDF doc encoding and is mapped to the Unicode value `2022`. Since `51` is `Q`, it is output as is. If you wanted to convert back from here to a binary string, would have to recognize Unicode values whose code points are higher than `0xFF` and map those back to their corresponding PDF doc encoding characters. There is no way to tell the difference between a Unicode string that was originally encoded as UTF-16 or one that was converted from PDF doc encoding. In other words, it's best if you don't try to use the JSON format to extract binary strings from the PDF file, but if you really had to, it could be done. Note that qpdf's **--show-object** option does not have this limitation and will reveal the string as encoded in the original file.

6.4. JSON: Special Considerations

For the most part, the built-in JSON help tells you everything you need to know about the JSON format, but there are a few non-obvious things to be aware of:

- While qpdf guarantees that keys present in the help will be present in the output, those fields may be null or empty if the information is not known or absent in the file. Also, if you specify **--json-keys**, the keys that are not listed will be excluded entirely except for those that **--json-help** says are always present.

- In a few places, there are keys with names containing `pageposfrom1`. The values of these keys are null or an integer. If an integer, they point to a page index within the file numbering from 1. Note that JSON indexes from 0, and you would also use 0-based indexing using the API. However, 1-based indexing is easier in this case because the command-line syntax for specifying page ranges is 1-based. If you were going to write a program that looked through the JSON for information about specific pages and then use the command-line to extract those pages, 1-based indexing is easier. Besides, it's more convenient to subtract 1 from a program in a real programming language than it is to add 1 from shell code.

- The image information included in the `page` section of the JSON output includes the key "`filterable`". Note that the value of this field may depend on the **--decode-level** that you invoke qpdf with. The JSON output includes a top-level key "`parameters`" that indicates the decode level used for computing whether a stream was filterable. For example, jpeg images will be shown as not filterable by default, but they will be shown as filterable if you run **qpdf --json --decode-level=all**.

Chapter 7. Design and Library Notes

7.1. 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.

7.2. 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, though it is possible to determine whether an object is direct or not if this information is needed. 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 C++-11's ***std::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.

The primary class for interacting with PDF objects is ***QPDFObjectHandle***. Instances of this class can be passed around by value, copied, stored in containers, etc. with very low overhead. Instances of ***QPDFObjectHandle*** created by reading from a file will 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 *QPDFObjec-*

tHandle::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.

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 files *QPDF.hh* and *QPDFObjectHandle.hh*. There are also some additional helper classes that provide higher level API functions for certain document constructions. These are discussed in Section 7.3, “Helper Classes”, page 34.

7.3. Helper Classes

QPDF version 8.1 introduced the concept of helper classes. Helper classes are intended to contain higher level APIs that allow developers to work with certain document constructs at an abstraction level above that of ***QPDFObjectHandle*** while staying true to qpdf's philosophy of not hiding document structure from the developer. As with qpdf in general, the goal is take away some of the more tedious bookkeeping aspects of working with PDF files, not to remove the need for the developer to understand how the PDF construction in question works. The driving factor behind the creation of helper classes was to allow the evolution of higher level interfaces in qpdf without polluting the interfaces of the main top-level classes ***QPDF*** and ***QPDFObjectHandle***.

There are two kinds of helper classes: *document* helpers and *object* helpers. Document helpers are constructed with a reference to a ***QPDF*** object and provide methods for working with structures that are at the document level. Object helpers are constructed with an instance of a ***QPDFObjectHandle*** and provide methods for working with specific types of objects.

Examples of document helpers include ***QPDFPageDocumentHelper***, which contains methods for operating on the document's page trees, such as enumerating all pages of a document and adding and removing pages; and ***QPDFAcroFormDocumentHelper***, which contains document-level methods related to interactive forms, such as enumerating form fields and creating mappings between form fields and annotations.

Examples of object helpers include ***QPDFPageObjectHelper*** for performing operations on pages such as page rotation and some operations on content streams, ***QPDFFormFieldObjectHelper*** for performing operations related to interactive form fields, and ***QPDFAnnotationObjectHelper*** for working with annotations.

It is always possible to retrieve the underlying ***QPDF*** reference from a document helper and the underlying ***QPDFObjectHandle*** reference from an object helper. Helpers are designed to be helpers, not wrappers. The intention is that, in general, it is safe to freely intermix operations that use helpers with operations that use the underlying objects. Document and object helpers do not attempt to provide a complete interface for working with the things they are

[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.

helping with, nor do they attempt to encapsulate underlying structures. They just provide a few methods to help with error-prone, repetitive, or complex tasks. In some cases, a helper object may cache some information that is expensive to gather. In such cases, the helper classes are implemented so that their own methods keep the cache consistent, and the header file will provide a method to invalidate the cache and a description of what kinds of operations would make the cache invalid. If in doubt, you can always discard a helper class and create a new one with the same underlying objects, which will ensure that you have discarded any stale information.

By Convention, document helpers are called ***QPDFSomethingDocumentHelper*** and are derived from ***QPDFDocumentHelper***, and object helpers are called ***QPDFSomethingObjectHelper*** and are derived from ***QPDFObjectHelper***. For details on specific helpers, please see their header files. You can find them by looking at *include/qpdf/QPDF*DocumentHelper.hh* and *include/qpdf/QPDF*ObjectHelper.hh*.

In order to avoid creation of circular dependencies, the following general guidelines are followed with helper classes:

- Core class interfaces do not know about helper classes. For example, no methods of ***QPDF*** or ***QPDFObjectHandle*** will include helper classes in their interfaces.
- Interfaces of object helpers will usually not use document helpers in their interfaces. This is because it is much more useful for document helpers to have methods that return object helpers. Most operations in PDF files start at the document level and go from there to the object level rather than the other way around. It can sometimes be useful to map back from object-level structures to document-level structures. If there is a desire to do this, it will generally be provided by a method in the document helper class.
- Most of the time, object helpers don't know about other object helpers. However, in some cases, one type of object may be a container for another type of object, in which case it may make sense for the outer object to know about the inner object. For example, there are methods in the ***QPDFPageObjectHelper*** that know ***QPDFAnnotationObjectHelper*** because references to annotations are contained in page dictionaries.
- Any helper or core library class may use helpers in their implementations.

Prior to qpdf version 8.1, higher level interfaces were added as “convenience functions” in either ***QPDF*** or ***QPDFObjectHandle***. For compatibility, older convenience functions for operating with pages will remain in those classes even as alternatives are provided in helper classes. Going forward, new higher level interfaces will be provided using helper classes.

7.4. Implementation Notes

This section contains a few notes about QPDF's internal implementation, particularly around what it does when it first processes a file. This section is a bit of a simplification of what it actually does, but it could serve as a starting point to someone trying to understand the implementation. There is nothing in this section that you need to know to use the qpdf library.

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***.

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 iterates 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.

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 a PDF header. 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.
- 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.

7.5. 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 ***QIntC*** namespace, provided by *include/qpdf/QIntC.hh*, implements safe functions for converting between integer types. These functions do range checking and throw a `std::range_error`, which is subclass of `std::runtime_error`, if conversion from one integer type to another results in loss of information. There are many cases in which we have to move between different integer types because of incompatible integer types used in interoperable interfaces. Some are unavoidable, such as moving between sizes and offsets, and others are there because of old code that is too in entrenched to be fixable without breaking source compatibility and causing pain for users. QPDF is compiled with extra warnings to detect conversions with potential data loss, and all such cases should be fixed by either using a function from ***QIntC*** or a *static_cast*.

When the intention is just to switch the type because of exchanging data between incompatible interfaces, use ***QIntC***. This is the usual case. However, there are some cases in which we are explicitly intending to use the exact same bit pattern with a different type. This is most common when switching between signed and unsigned characters. A lot of qpdf's code uses unsigned characters internally, but `std::string` and `char` are signed. Using *QIntC::to_char* would be wrong for converting from unsigned to signed characters because a negative `char` value and the corresponding `unsigned char` value greater than 127 *mean the same thing*. There are also cases in which we use *static_cast* when working with bit fields where we are not representing a numerical value but rather a bunch of bits packed together in some integer type. Also note that `size_t` and `long` both typically differ between 32-bit and 64-bit environments, so sometimes an explicit cast may not be needed to avoid warnings on one platform but may be needed on another. A conversion with ***QIntC*** should always be used when the types are different even if the underlying size is the same. QPDF's CI build builds on 32-bit and 64-bit platforms, and the test suite is very thorough, so it is hard to make any of the potential errors here without being caught in build or test.

Non-const `unsigned char*` is 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 `Pipeline`s. This could theoretically be unsafe, but there is adequate testing to assert that it is safe and will remain safe in qpdf's code.

7.6. 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.

One point of confusion some people have about encrypted PDF files is that encryption is not the same as password protection. Password protected files are always encrypted, but it is also possible to create encrypted files that do not have passwords. Internally, such files use the empty string as a password, and most readers try the empty string first to see if it works and prompt for a password only if the empty string doesn't work. Normally such files have an empty user password and a non-empty owner password. In that way, if the file is opened by an ordinary reader without specification of password, the restrictions specified in the encryption dictionary can be enforced. Most users wouldn't even realize such a file was encrypted. Since qpdf always ignores the restrictions (except for the purpose of reporting what they are), qpdf doesn't care which password you use. QPDF will allow you to create PDF files with non-empty user passwords and empty owner passwords. Some readers will require a password when you open these files, and others will open the files without a password and not enforce restrictions. Having a non-empty user password and an empty owner password doesn't really make sense because it would mean that opening the file with the user password would be more restrictive than not supplying a password at all. QPDF also allows you to create PDF files with the same password as both the user and owner password. Some readers will not ever allow such files to be accessed without restrictions because they never try the password as the owner password if it works as the user password. Nonetheless, one of the powerful aspects of qpdf is that it allows you to finely specify the way encrypted files are created, even if the results are not useful to some readers. One use case for this would be for testing a PDF reader to ensure that it handles odd configurations of input files.

7.7. Random Number Generation

QPDF generates random numbers to support generation of encrypted data. Starting in qpdf 10.0.0, qpdf uses the crypto provider as its source of random numbers. Older versions used the OS-provided source of secure random numbers or, if allowed at build time, insecure random numbers from stdlib. Starting with version 5.1.0, you can disable use of OS-provided secure random numbers at build time. This is especially useful on Windows if you want to avoid a dependency on Microsoft's cryptography API. You can also supply your own random data provider. For details on how to do this, please refer to the top-level README.md file in the source distribution and to comments in *QUtil.hh*.

7.8. 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*.

7.9. 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 Section 7.10, “Copying Objects From Other PDF Files”, page 39. For an example of how to use reserved objects, search for *newReserved* in *test_driver.cc* in qpdf's sources.

7.10. 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 “from” ***QPDF*** object must remain valid after the copy as discussed in the note below. 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.

Please note: when you copy objects from one ***QPDF*** to another, the source ***QPDF*** object must remain valid until you have finished with the destination object. This is because the original object is still used to retrieve any referenced stream data from the copied object.

7.11. 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 8, *Linearization*, page 41 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 26.

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`.

7.12. 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 8. Linearization

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

8.1. 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.

8.2. 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 8.3, “Optimization”, page 41. 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

8.3. 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.

8.4. 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.

8.5. 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.)

8.6. 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.

8.7. 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 9. Object and Cross-Reference Streams

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

9.1. 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 9.3, “Implications for Linearized Files”, page 45for 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.

9.2. 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.

9.2.1. 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.

9.3. 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.

9.4. 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.

10.1.0: January 5, 2021

- CLI Enhancements
  - Add **--flatten-rotation** command-line option, which causes all pages that are rotated using parameters in the page's dictionary to instead be identically rotated in the page's contents. The change is not user-visible for compliant PDF readers but can be used to work around broken PDF applications that don't properly handle page rotation.
- Library Enhancements
  - Support for user-provided (pluggable, modular) stream filters. It is now possible to derive a class from ***QPDFStreamFilter*** and register it with ***QPDF*** so that regular library methods, including those used by ***QPDFWriter***, can decode streams with filters not directly supported by the library. The example *examples/pdf-custom-filter.cc* illustrates how to use this capability.
  - Add methods to ***QPDFPageObjectHelper*** to iterate through XObjects on a page or form XObjects, possibly recursing into nested form XObjects: *forEachXObject*, *ForEachImage*, *forEachFormXObject*.
  - Enhance several methods in ***QPDFPageObjectHelper*** to work with form XObjects as well as pages, as noted in comments. See *ChangeLog* for a full list.
  - Rename some functions in ***QPDFPageObjectHelper***, while keeping old names for compatibility:
    - *getPageImages* to *getImages*
    - *filterPageContents* to *filterContents*
    - *pipePageContents* to *pipeContents*
    - *parsePageContents* to *parseContents*
  - Add method *QPDFPageObjectHelper::getFormXObjects* to return a map of form XObjects directly on a page or form XObject
  - Add new helper methods to ***QPDFObjectHandle***: *isFormXObject*, *isImage*
  - Add the optional *allow_streams* parameter *QPDFObjectHandle::makeDirect*. When *QPDFObjectHandle::makeDirect* is called in this way, it preserves references to streams rather than throwing an exception.
  - Add *QPDFObjectHandle::setFilterOnWrite* method. Calling this on a stream prevents *QPDFWriter* from attempting to uncompress, recompress, or otherwise filter a stream even if it could. Developers can use this to protect streams that are optimized should be protected from ***QPDFWriter***'s default behavior for any other reason.
  - Add ***ostream*** **<<** operator for ***QPDFObjGen***. This is useful to have for debugging.
  - Add method *QPDFPageObjectHelper::flattenRotation*, which replaces a page's `/Rotate` keyword by rotating the page within the content stream and altering the page's bounding boxes so the rendering is the same. This can be used to work around buggy PDF readers that can't properly handle page rotation.

- C API Enhancements
  - Add several new functions to the C API for working with objects. These are wrappers around many of the methods in ***QPDFObjectHandle***. Their inclusion adds considerable new capability to the C API.
  - Add *qpdf_register_progress_reporter* to the C API, corresponding to *QPDFWriter::registerProgressReporter*.
- Performance Enhancements
  - Improve steps ***QPDFWriter*** takes to prepare a ***QPDF*** object for writing, resulting in about an 8% improvement in write performance while allowing indirect objects to appear in `/DecodeParms`.
  - When extracting pages, the **qpdf** CLI only removes unreferenced resources from the pages that are being kept, resulting in a significant performance improvement when extracting small numbers of pages from large, complex documents.
- Bug Fixes
  - *QPDFPageObjectHelper::externalizeInlineImages* was not externalizing images referenced from form XObjects that appeared on the page.
  - *QPDFObjectHandle::filterPageContents* was broken for pages with multiple content streams.
  - Tweak zsh completion code to behave a little better with respect to path completion.

10.0.4: November 21, 2020

- Bug Fixes
  - Fix a handful of integer overflows. This includes cases found by fuzzing as well as having qpdf not do range checking on unused values in the xref stream.

10.0.3: October 31, 2020

- Bug Fixes
  - The fix to the bug involving copying streams with indirect filters was incorrect and introduced a new, more serious bug. The original bug has been fixed correctly, as has the bug introduced in 10.0.2.

10.0.2: October 27, 2020

- Bug Fixes
  - When concatenating content streams, as with **--coalesce-contents**, there were cases in which qpdf would merge two lexical tokens together, creating invalid results. A newline is now inserted between merged content streams if one is not already present.
  - Fix an internal error that could occur when copying foreign streams whose stream data had been replaced using a stream data provider if those streams had indirect filters or decode parameters. This is a rare corner case.
  - Ensure that the caller's locale settings do not change the results of numeric conversions performed internally by the qpdf library. Note that the problem here could only be caused when the qpdf library was used programmatically. Using the qpdf CLI already ignored the user's locale for numeric conversion.
  - Fix several instances in which warnings were not suppressed in spite of **--no-warn** and/or errors or warnings were written to standard output rather than standard error.

- Fixed a memory leak that could occur under specific circumstances when **--object-streams=generate** was used.
- Fix various integer overflows and similar conditions found by the OSS-Fuzz project.

- Enhancements
  - New option **--warning-exit-0** causes qpdf to exit with a status of `0` rather than `3` if there are warnings but no errors. Combine with **--no-warn** to completely ignore warnings.
  - Performance improvements have been made to *QPDF::processMemoryFile*.
  - The OpenSSL crypto provider produces more detailed error messages.
- Build Changes
  - The option **--disable-rpath** is now supported by qpdf's **./configure** script. Some distributions' packaging standards recommended the use of this option.
  - Selection of a printf format string for `long long` has been moved from `ifdefs` to an autoconf test. If you are using your own build system, you will need to provide a value for `LL_FMT` in *libqpdf/qpdf/qpdf-config.h*, which would typically be `"%lld"` or, for some Windows compilers, `"%I64d"`.
  - Several improvements were made to build-time configuration of the OpenSSL crypto provider.
  - A nearly stand-alone Linux binary zip file is now included with the qpdf release. This is built on an older (but supported) Ubuntu LTS release, but would work on most reasonably recent Linux distributions. It contains only the executables and required shared libraries that would not be present on a minimal system. It can be used for including qpdf in a minimal environment, such as a docker container. The zip file is also known to work as a layer in AWS Lambda.
  - QPDF's automated build has been migrated from Azure Pipelines to GitHub Actions.
- Windows-specific Changes
  - The Windows executables distributed with qpdf releases now use the OpenSSL crypto provider by default. The native crypto provider is also compiled in and can be selected at runtime with the `QPDF_CRYPTO_PROVIDER` environment variable.
  - Improvements have been made to how a cryptographic provider is obtained in the native Windows crypto implementation. However mostly this is shadowed by OpenSSL being used by default.

10.0.1: April 9, 2020

- Bug Fixes
  - 10.0.0 introduced a bug in which calling *QPDFObjectHandle::getStreamData* on a stream that can't be filtered was returning the raw data instead of throwing an exception. This is now fixed.
  - Fix a bug that was preventing qpdf from linking with some versions of clang on some platforms.
- Enhancements
  - Improve the *pdf-invert-images* example to avoid having to load all the images into RAM at the same time.

10.0.0: April 6, 2020

- Performance Enhancements

  - The qpdf library and executable should run much faster in this version than in the last several releases. Several internal library optimizations have been made, and there has been improved behavior on page splitting as well. This version of qpdf should outperform any of the 8.x or 9.x versions.

- Incompatible API (source-level) Changes (minor)

  - The *QUtil::srandom* method was removed. It didn't do anything unless insecure random numbers were compiled in, and they have been off by default for a long time. If you were calling it, just remove the call since it wasn't doing anything anyway.

- Build/Packaging Changes

  - Add a `openssl` crypto provider, which is implemented with OpenSSL and also works with BoringSSL. Thanks to Dean Scarff for this contribution. If you maintain qpdf for a distribution, pay special attention to make sure that you are including support for the crypto providers you want. Package maintainers will have to weigh the advantages of allowing users to pick a crypto provider at runtime against the disadvantages of adding more dependencies to qpdf.

  - Allow qpdf to built on stripped down systems whose C/C++ libraries lack the ***wchar_t*** type. Search for ***wchar_t*** in qpdf's README.md for details. This should be very rare, but it is known to be helpful in some embedded environments.

- CLI Enhancements

  - Add `objectinfo` key to the JSON output. This will be a place to put computed metadata or other information about PDF objects that are not immediately evident in other ways or that seem useful for some other reason. In this version, information is provided about each object indicating whether it is a stream and, if so, what its length and filters are. Without this, it was not possible to tell conclusively from the JSON output alone whether or not an object was a stream. Run **qpdf --json-help** for details.

  - Add new option **--remove-unreferenced-resources** which takes `auto`, `yes`, or `no` as arguments. The new `auto` mode, which is the default, performs a fast heuristic over a PDF file when splitting pages to determine whether the expensive process of finding and removing unreferenced resources is likely to be of benefit. For most files, this new default will result in a significant performance improvement for splitting pages. See Section 3.8, “Advanced Transformation Options”, page 17 for a more detailed discussion.

  - The **--preserve-unreferenced-resources** is now just a synonym for **--remove-unreferenced-resources=no**.

  - If the `QPDF_EXECUTABLE` environment variable is set when invoking **qpdf --bash-completion** or **qpdf --zsh-completion**, the completion command that it outputs will refer to qpdf using the value of that variable rather than what **qpdf** determines its executable path to be. This can be useful when wrapping **qpdf** with a script, working with a version in the source tree, using an AppImage, or other situations where there is some indirection.

- Library Enhancements

  - Random number generation is now delegated to the crypto provider. The old behavior is still used by the native crypto provider. It is still possible to provide your own random number generator.

  - Add a new version of *QPDFObjectHandle::StreamDataProvider::provideStreamData* that accepts the *suppress_warnings* and *will_retry* options and allows a success code to be returned. This makes it possible to implement a ***StreamDataProvider*** that calls *pipeStreamData* on another stream and to pass the response back to the caller, which enables better error handling on those proxied streams.

  - Update *QPDFObjectHandle::pipeStreamData* to return an overall success code that goes beyond whether or not filtered data was written successfully. This allows better error handling of cases that were not filtering

errors. You have to call this explicitly. Methods in previously existing APIs have the same semantics as before.

- The *QPDFPageObjectHelper::placeFormXObject* method now allows separate control over whether it should be willing to shrink or expand objects to fit them better into the destination rectangle. The previous behavior was that shrinking was allowed but expansion was not. The previous behavior is still the default.

- When calling the C API, any non-zero value passed to a boolean parameter is treated as `TRUE`. Previously only the value `1` was accepted. This makes the C API behave more like most C interfaces and is known to improve compatibility with some Windows environments that dynamically load the DLL and call functions from it.

- Add *QPDFObjectHandle::unsafeShallowCopy* for copying only top-level dictionary keys or array items. This is unsafe because it creates a situation in which changing a lower-level item in one object may also change it in another object, but for cases in which you *know* you are only inserting or replacing top-level items, it is much faster than *QPDFObjectHandle::shallowCopy*.

- Add *QPDFObjectHandle::filterAsContents*, which filter's a stream's data as a content stream. This is useful for parsing the contents for form XObjects in the same way as parsing page content streams.

- Bug Fixes

  - When detecting and removing unreferenced resources during page splitting, traverse into form XObjects and handle their resources dictionaries as well.

  - The same error recovery is applied to streams in other than the primary input file when merging or splitting pages.

9.1.1: January 26, 2020

- Build/Packaging Changes

  - The fix-qdf program was converted from perl to C++. As such, qpdf no longer has a runtime dependency on perl.

- Library Enhancements

  - Added new helper routine *QUtil::call_main_from_wmain* which converts `wchar_t` arguments to UTF-8 encoded strings. This is useful for qpdf because library methods expect file names to be UTF-8 encoded, even on Windows

  - Added new *QUtil::read_lines_from_file* methods that take `FILE*` arguments and that allow preservation of end-of-line characters. This also fixes a bug where *QUtil::read_lines_from_file* wouldn't work properly with Unicode filenames.

- CLI Enhancements

  - Added options **--is-encrypted** and **--requires-password** for testing whether a file is encrypted or requires a password other than the supplied (or empty) password. These communicate via exit status, making them useful for shell scripts. They also work on encrypted files with unknown passwords.

  - Added `encrypt` key to JSON options. With the exception of the reconstructed user password for older encryption formats, this provides the same information as **--show-encryption** but in a consistent, parseable format. See output of **qpdf --json-help** for details.

- Bug Fixes

- In QDF mode, be sure not to write more than one XRef stream to a file, even when **--preserve-unreferenced** is used. **fix-qdf** assumes that there is only one XRef stream, and that it appears at the end of the file.
- When externalizing inline images, properly handle images whose color space is a reference to an object in the page's resource dictionary.
- Windows-specific fix for acquiring crypt context with a new keyset.

9.1.0: November 17, 2019

- Build Changes
  - A C++-11 compiler is now required to build qpdf.
  - A new crypto provider that uses gnutls for crypto functions is now available and can be enabled at build time. See Section 2.3, “Crypto Providers”, page 3 for more information about crypto providers and Section 2.3.1, “Build Support For Crypto Providers”, page 3 for specific information about the build.
- Library Enhancements
  - Incorporate contribution from Masamichi Hosoda to properly handle signature dictionaries by not including them in object streams, formatting the `Contents` key has a hexadecimal string, and excluding the `/Contents` key from encryption and decryption.
  - Incorporate contribution from Masamichi Hosoda to provide new API calls for getting file-level information about input and output files, enabling certain operations on the files at the file level rather than the object level. New methods include *QPDF::getXRefTable()*, *QPDFObjectHandle::getParsedOffset()*, *QPDFWriter::getRenumberedObjGen(QPDFObjGen)*, and *QPDFWriter::getWrittenXRefTable()*.
  - Support build-time and runtime selectable crypto providers. This includes the addition of new classes ***QPDFCryptoProvider*** and ***QPDFCryptoImpl*** and the recognition of the `QPDF_CRYPTO_PROVIDER` environment variable. Crypto providers are described in depth in Section 2.3, “Crypto Providers”, page 3.
- CLI Enhancements
  - Addition of the **--show-crypto** option in support of selectable crypto providers, as described in Section 2.3, “Crypto Providers”, page 3.
  - Allow `:even` or `:odd` to be appended to numeric ranges for specification of the even or odd pages from among the pages specified in the range.
  - Fix shell wildcard expansion behavior (`*` and `?`) of the **qpdf.exe** as built my MSVC.

9.0.2: October 12, 2019

- Bug Fix
  - Fix the name of the temporary file used by **--replace-input** so that it doesn't require path splitting and works with paths include directories.

9.0.1: September 20, 2019

- Bug Fixes/Enhancements
  - Fix some build and test issues on big-endian systems and compilers with characters that are unsigned by default. The problems were in build and test only. There were no actual bugs in the qpdf library itself relating to endianness or unsigned characters.

- When a dictionary has a duplicated key, report this with a warning. The behavior of the library in this case is unchanged, but the error condition is no longer silently ignored.

- When a form field's display rectangle is erroneously specified with inverted coordinates, detect and correct this situation. This avoids some form fields from being flipped when flattening annotations on files with this condition.

9.0.0: August 31, 2019

- Incompatible API (source-level) Changes (minor)

  - The method *QUtil::strcasecmp* has been renamed to *QUtil::str_compare_nocase*. This incompatible change is necessary to enable qpdf to build on platforms that define *strcasecmp* as a macro.

  - The *QPDF::copyForeignObject* method had an overloaded version that took a boolean parameter that was not used. If you were using this version, just omit the extra parameter.

  - There was a version *QPDFTokenizer::expectInlineImage* that took no arguments. This version has been removed since it caused the tokenizer to return incorrect inline images. A new version was added some time ago that produces correct output. This is a very low level method that doesn't make sense to call outside of qpdf's lexical engine. There are higher level methods for tokenizing content streams.

  - Change *QPDFOutlineDocumentHelper::getTopLevelOutlines* and *QPDFOutlineObjectHelper::getKids* to return a `std::vector` instead of a `std::list` of ***QPDFOutlineObjectHelper*** objects.

  - Remove method *QPDFTokenizer::allowPoundAnywhereInName*. This function would allow creation of name tokens whose value would change when unparsed, which is never the correct behavior.

- CLI Enhancements

  - The **--replace-input** option may be given in place of an output file name. This causes qpdf to overwrite the input file with the output. See the description of **--replace-input** in Section 3.3, “Basic Options”, page 7 for more details.

  - The **--recompress-flate** instructs **qpdf** to recompress streams that are already compressed with `/FlateDecode`. Useful with **--compression-level**.

  - The **--compression-level=*level*** sets the zlib compression level used for any streams compressed by `/FlateDecode`. Most effective when combined with **--recompress-flate**.

- Library Enhancements

  - A new namespace ***QIntC***, provided by *qpdf/QIntC.hh*, provides safe conversion methods between different integer types. These conversion methods do range checking to ensure that the cast can be performed with no loss of information. Every use of *static_cast* in the library was inspected to see if it could use one of these safe converters instead. See Section 7.5, “Casting Policy”, page 36 for additional details.

  - Method *QPDF::anyWarnings* tells whether there have been any warnings without clearing the list of warnings.

  - Method *QPDF::closeInputSource* closes or otherwise releases the input source. This enables the input file to be deleted or renamed.

  - New methods have been added to ***QUtil*** for converting back and forth between strings and unsigned integers: *uint_to_string*, *uint_to_string_base*, *string_to_uint*, and *string_to_ull*.

- New methods have been added to ***QPDFObjectHandle*** that return the value of ***Integer*** objects as `int` or `unsigned int` with range checking and sensible fallback values, and a new method was added to return an unsigned value. This makes it easier to write code that is safe from unintentional data loss. Functions: *getUIntValue*, *getIntVauleAsInt*, *getUIntValueAsUInt*.

- When parsing content streams with ***QPDFObjectHandle::ParserCallbacks***, in place of the method *handleObject(QPDFObjectHandle)*, the developer may override *handleObject(QPDFObjectHandle, size_t offset, size_t length)*. If this method is defined, it will be invoked with the object along with its offset and length within the overall contents being parsed. Intervening spaces and comments are not included in offset and length. Additionally, a new method *contentSize(size_t)* may be implemented. If present, it will be called prior to the first call to *handleObject* with the total size in bytes of the combined contents.

- New methods *QPDF::userPasswordMatched* and *QPDF::ownerPasswordMatched* have been added to enable a caller to determine whether the supplied password was the user password, the owner password, or both. This information is also displayed by **qpdf --show-encryption** and **qpdf --check**.

- Static method *Pl_Flate::setCompressionLevel* can be called to set the zlib compression level globally used by all instances of Pl_Flate in deflate mode.

- The method *QPDFWriter::setRecompressFlate* can be called to tell ***QPDFWriter*** to uncompress and recompress streams already compressed with `/FlateDecode`.

- The underlying implementation of QPDF arrays has been enhanced to be much more memory efficient when dealing with arrays with lots of nulls. This enables qpdf to use drastically less memory for certain types of files.

- When traversing the pages tree, if nodes are encountered with invalid types, the types are fixed, and a warning is issued.

- A new helper method *QUtil::read_file_into_memory* was added.

- All conditions previously reported by *QPDF::checkLinearization()* as errors are now presented as warnings.

- Name tokens containing the # character not preceded by two hexadecimal digits, which is invalid in PDF 1.2 and above, are properly handled by the library: a warning is generated, and the name token is properly preserved, even if invalid, in the output. See *ChangeLog* for a more complete description of this change.

- Bug Fixes

  - A small handful of memory issues, assertion failures, and unhandled exceptions that could occur on badly mangled input files have been fixed. Most of these problems were found by Google's OSS-Fuzz project.

  - When **qpdf --check** or **qpdf --check-linearization** encounters a file with linearization warnings but not errors, it now properly exits with exit code 3 instead of 2.

  - The **--completion-bash** and **--completion-zsh** options now work properly when qpdf is invoked as an AppImage.

  - Calling *QPDFWriter::set*EncryptionParameters* on a ***QPDFWriter*** object whose output filename has not yet been set no longer produces a segmentation fault.

  - When reading encrypted files, follow the spec more closely regarding encryption key length. This allows qpdf to open encrypted files in most cases when they have invalid or missing /Length keys in the encryption dictionary.

- Build Changes

- On platforms that support it, qpdf now builds with **-fvisibility=hidden**. If you build qpdf with your own build system, this is now safe to use. This prevents methods that are not part of the public API from being exported by the shared library, and makes qpdf's ELF shared libraries (used on Linux, MacOS, and most other UNIX flavors) behave more like the Windows DLL. Since the DLL already behaves in much this way, it is unlikely that there are any methods that were accidentally not exported. However, with ELF shared libraries, typeinfo for some classes has to be explicitly exported. If there are problems in dynamically linked code catching exceptions or subclassing, this could be the reason. If you see this, please report a bug at https://github.com/qpdf/qpdf/issues/.

- QPDF is now compiled with integer conversion and sign conversion warnings enabled. Numerous changes were made to the library to make this safe.

- QPDF's **make install** target explicitly specifies the mode to use when installing files instead of relying the user's umask. It was previously doing this for some files but not others.

- If **pkg-config** is available, use it to locate *libjpeg* and *zlib* dependencies, falling back on old behavior if unsuccessful.

- Other Notes

  - QPDF has been fully integrated into Google's OSS-Fuzz project [https://github.com/google/oss-fuzz]. This project exercises code with randomly mutated inputs and is great for discovering hidden security crashes and security issues. Several bugs found by oss-fuzz have already been fixed in qpdf.

8.4.2: May 18, 2019

This release has just one change: correction of a buffer overrun in the Windows code used to open files. Windows users should take this update. There are no code changes that affect non-Windows releases.

8.4.1: April 27, 2019

- Enhancements

  - When **qpdf --version** is run, it will detect if the qpdf CLI was built with a different version of qpdf than the library, which may indicate a problem with the installation.

  - New option **--remove-page-labels** will remove page labels before generating output. This used to happen if you ran **qpdf --empty --pages .. --**, but the behavior changed in qpdf 8.3.0. This option enables people who were relying on the old behavior to get it again.

  - New option **--keep-files-open-threshold=*count*** can be used to override number of files that qpdf will use to trigger the behavior of not keeping all files open when merging files. This may be necessary if your system allows fewer than the default value of 200 files to be open at the same time.

- Bug Fixes

  - Handle Unicode characters in filenames on Windows. The changes to support Unicode on the CLI in Windows broke Unicode filenames for Windows.

  - Slightly tighten logic that determines whether an object is a page. This should resolve problems in some rare files where some non-page objects were passing qpdf's test for whether something was a page, thus causing them to be erroneously lost during page splitting operations.

  - Revert change that included preservation of outlines (bookmarks) in **--split-pages**. The way it was implemented in 8.3.0 and 8.4.0 caused a very significant degradation of performance for splitting certain files. A future release of qpdf may re-introduce the behavior in a more performant and also more correct fashion.

- In JSON mode, add missing leading 0 to decimal values between -1 and 1 even if not present in the input. The JSON specification requires the leading 0. The PDF specification does not.

8.4.0: February 1, 2019

- Command-line Enhancements
  - *Non-compatible CLI change:* The qpdf command-line tool interprets passwords given at the command-line differently from previous releases when the passwords contain non-ASCII characters. In some cases, the behavior differs from previous releases. For a discussion of the current behavior, please see Section 3.10, “Unicode Passwords”, page 24. The incompatibilities are as follows:
    - On Windows, qpdf now receives all command-line options as Unicode strings if it can figure out the appropriate compile/link options. This is enabled at least for MSVC and mingw builds. That means that if non-ASCII strings are passed to the qpdf CLI in Windows, qpdf will now correctly receive them. In the past, they would have either been encoded as Windows code page 1252 (also known as “Windows ANSI” or as something unintelligible. In almost all cases, qpdf is able to properly interpret Unicode arguments now, whereas in the past, it would almost never interpret them properly. The result is that non-ASCII passwords given to the qpdf CLI on Windows now have a much greater chance of creating PDF files that can be opened by a variety of readers. In the past, usually files encrypted from the Windows CLI using non-ASCII passwords would not be readable by most viewers. Note that the current version of qpdf is able to decrypt files that it previously created using the previously supplied password.
    - The PDF specification requires passwords to be encoded as UTF-8 for 256-bit encryption and with PDF Doc encoding for 40-bit or 128-bit encryption. Older versions of qpdf left it up to the user to provide passwords with the correct encoding. The qpdf CLI now detects when a password is given with UTF-8 encoding and automatically transcodes it to what the PDF spec requires. While this is almost always the correct behavior, it is possible to override the behavior if there is some reason to do so. This is discussed in more depth in Section 3.10, “Unicode Passwords”, page 24.
  - New options **--externalize-inline-images**, **--ii-min-bytes**, and **--keep-inline-images** control qpdf's handling of inline images and possible conversion of them to regular images. By default, **--optimize-images** now also applies to inline images. These options are discussed in Section 3.8, “Advanced Transformation Options”, page 17.
  - Add options **--overlay** and **--underlay** for overlaying or underlaying pages of other files onto output pages. See Section 3.6, “Overlay and Underlay Options”, page 15 for details.
  - When opening an encrypted file with a password, if the specified password doesn't work and the password contains any non-ASCII characters, qpdf will try a number of alternative passwords to try to compensate for possible character encoding errors. This behavior can be suppressed with the **--suppress-password-recovery** option. See Section 3.10, “Unicode Passwords”, page 24 for a full discussion.
  - Add the **--password-mode** option to fine-tune how qpdf interprets password arguments, especially when they contain non-ASCII characters. See Section 3.10, “Unicode Passwords”, page 24 for more information.
  - In the **--pages** option, it is now possible to copy the same page more than once from the same file without using the previous workaround of specifying two different paths to the same file.
  - In the **--pages** option, allow use of “.” as a shortcut for the primary input file. That way, you can do **qpdf in.pdf --pages . 1-2 -- out.pdf** instead of having to repeat *in.pdf* in the command.
  - When encrypting with 128-bit and 256-bit encryption, new encryption options **--assemble**, **--annotate**, **--form**, and **--modify-other** allow more fine-grained granularity in configuring options. Before, the **--modify** option only configured certain predefined groups of permissions.

- Bug Fixes and Enhancements
  - *Potential data-loss bug:* Versions of qpdf between 8.1.0 and 8.3.0 had a bug that could cause page splitting and merging operations to drop some font or image resources if the PDF file's internal structure shared these resource lists across pages and if some but not all of the pages in the output did not reference all the fonts and images. Using the **--preserve-unreferenced-resources** option would work around the incorrect behavior. This bug was the result of a typo in the code and a deficiency in the test suite. The case that triggered the error was known, just not handled properly. This case is now exercised in qpdf's test suite and properly handled.
  - When optimizing images, detect and refuse to optimize images that can't be converted to JPEG because of bit depth or color space.
  - Linearization and page manipulation APIs now detect and recover from files that have duplicate Page objects in the pages tree.
  - Using older option **--stream-data=compress** with object streams, object streams and xref streams were not compressed.
  - When the tokenizer returns inline image tokens, delimiters following `ID` and `EI` operators are no longer excluded. This makes it possible to reliably extract the actual image data.
- Library Enhancements
  - Add method *QPDFPageObjectHelper::externalizeInlineImages* to convert inline images to regular images.
  - Add method *QUtil::possible_repaired_encodings()* to generate a list of strings that represent other ways the given string could have been encoded. This is the method the QPDF CLI uses to generate the strings it tries when recovering incorrectly encoded Unicode passwords.
  - Add new versions of *QPDFWriter::setR{3,4,5,6}EncryptionParameters* that allow more granular setting of permissions bits. See *QPDFWriter.hh* for details.
  - Add new versions of the transcoders from UTF-8 to single-byte coding systems in ***QUtil*** that report success or failure rather than just substituting a specified unknown character.
  - Add method *QUtil::analyze_encoding()* to determine whether a string has high-bit characters and is appears to be UTF-16 or valid UTF-8 encoding.
  - Add new method *QPDFPageObjectHelper::shallowCopyPage()* to copy a new page that is a “shallow copy” of a page. The resulting object is an indirect object ready to be passed to *QPDFPageDocumentHelper::addPage()* for either the original ***QPDF*** object or a different one. This is what the **qpdf** command-line tool uses to copy the same page multiple times from the same file during splitting and merging operations.
  - Add method *QPDF::getUniqueId()*, which returns a unique identifier for the given QPDF object. The identifier will be unique across the life of the application. The returned value can be safely used as a map key.
  - Add method *QPDF::setImmediateCopyFrom*. This further enhances qpdf's ability to allow a ***QPDF*** object from which objects are being copied to go out of scope before the destination object is written. If you call this method on a ***QPDF*** instances, objects copied *from* this instance will be copied immediately instead of lazily. This option uses more memory but allows the source object to go out of scope before the destination object is written in all cases. See comments in *QPDF.hh* for details.
  - Add method *QPDFPageObjectHelper::getAttribute* for retrieving an attribute from the page dictionary taking inheritance into consideration, and optionally making a copy if your intention is to modify the attribute.

- Fix long-standing limitation of *QPDFPageObjectHelper::getPageImages* so that it now properly reports images from inherited resources dictionaries, eliminating the need to call *QPDFPageDocumentHelper::pushInheritedAttributesToPage* in this case.

- Add method *QPDFObjectHandle::getUniqueResourceName* for finding an unused name in a resource dictionary.

- Add method *QPDFPageObjectHelper::getFormXObjectForPage* for generating a form XObject equivalent to a page. The resulting object can be used in the same file or copied to another file with *copyForeignObject*. This can be useful for implementing underlay, overlay, n-up, thumbnails, or any other functionality requiring replication of pages in other contexts.

- Add method *QPDFPageObjectHelper::placeFormXObject* for generating content stream text that places a given form XObject on a page, centered and fit within a specified rectangle. This method takes care of computing the proper transformation matrix and may optionally compensate for rotation or scaling of the destination page.

- Build Improvements

  - Add new configure option **--enable-avoid-windows-handle**, which causes the preprocessor symbol `AVOID_WINDOWS_HANDLE` to be defined. When defined, qpdf will avoid referencing the Windows ***HANDLE*** type, which is disallowed with certain versions of the Windows SDK.

  - For Windows builds, attempt to determine what options, if any, have to be passed to the compiler and linker to enable use of *wmain*. This causes the preprocessor symbol `WINDOWS_WMAIN` to be defined. If you do your own builds with other compilers, you can define this symbol to cause *wmain* to be used. This is needed to allow the Windows **qpdf** command to receive Unicode command-line options.

8.3.0: January 7, 2019

- Command-line Enhancements

  - Shell completion: you can now use eval **$(qpdf --completion-bash)** and eval **$(qpdf --completion-zsh)** to enable shell completion for bash and zsh.

  - Page numbers (also known as page labels) are now preserved when merging and splitting files with the **--pages** and **--split-pages** options.

  - Bookmarks are partially preserved when splitting pages with the **--split-pages** option. Specifically, the outlines dictionary and some supporting metadata are copied into the split files. The result is that all bookmarks from the original file appear, those that point to pages that are preserved work, and those that point to pages that are not preserved don't do anything. This is an interim step toward proper support for bookmarks in splitting and merging operations.

  - Page collation: add new option **--collate**. When specified, the semantics of **--pages** change from concatenation to collation. See Section 3.5, “Page Selection Options”, page 13 for examples and discussion.

  - Generation of information in JSON format, primarily to facilitate use of qpdf from languages other than C ++. Add new options **--json**, **--json-key**, and **--json-object** to generate a JSON representation of the PDF file. Run **qpdf --json-help** to get a description of the JSON format. For more information, see Chapter 6, *QPDF JSON*, page 30.

  - The **--generate-appearances** flag will cause qpdf to generate appearances for form fields if the PDF file indicates that form field appearances are out of date. This can happen when PDF forms are filled in by a program that doesn't know how to regenerate the appearances of the filled-in fields.

- The **--flatten-annotations** flag can be used to *flatten* annotations, including form fields. Ordinarily, annotations are drawn separately from the page. Flattening annotations is the process of combining their appearances into the page's contents. You might want to do this if you are going to rotate or combine pages using a tool that doesn't understand about annotations. You may also want to use **--generate-appearances** when using this flag since annotations for outdated form fields are not flattened as that would cause loss of information.

- The **--optimize-images** flag tells qpdf to recompresses every image using DCT (JPEG) compression as long as the image is not already compressed with lossy compression and recompressing the image reduces its size. The additional options **--oi-min-width**, **--oi-min-height**, and **--oi-min-area** prevent recompression of images whose width, height, or pixel area (width × height) are below a specified threshold.

- The **--show-object** option can now be given as **--show-object=trailer** to show the trailer dictionary.

- Bug Fixes and Enhancements

  - QPDF now automatically detects and recovers from dangling references. If a PDF file contained an indirect reference to a non-existent object, which is valid, when adding a new object to the file, it was possible for the new object to take the object ID of the dangling reference, thereby causing the dangling reference to point to the new object. This case is now prevented.

  - Fixes to form field setting code: strings are always written in UTF-16 format, and checkboxes and radio buttons are handled properly with respect to synchronization of values and appearance states.

  - The *QPDF::checkLinearization()* no longer causes the program to crash when it detects problems with linearization data. Instead, it issues a normal warning or error.

  - Ordinarily qpdf treats an argument of the form **@file** to mean that command-line options should be read from *file*. Now, if *file* does not exist but *@file* does, qpdf will treat *@file* as a regular option. This makes it possible to work more easily with PDF files whose names happen to start with the `@` character.

- Library Enhancements

  - Remove the restriction in most cases that the source QPDF object used in a *QPDF::copyForeignObject* call has to stick around until the destination QPDF is written. The exceptional case is when the source stream gets is data using a QPDFObjectHandle::StreamDataProvider. For a more in-depth discussion, see comments around *copyForeignObject* in *QPDF.hh*.

  - Add new method *QPDFWriter::getFinalVersion()*, which returns the PDF version that will ultimately be written to the final file. See comments in *QPDFWriter.hh* for some restrictions on its use.

  - Add several methods for transcoding strings to some of the character sets used in PDF files: *QUtil::utf8_to_ascii*, *QUtil::utf8_to_win_ansi*, *QUtil::utf8_to_mac_roman*, and *QUtil::utf8_to_utf16*. For the single-byte encodings that support only a limited character sets, these methods replace unsupported characters with a specified substitute.

  - Add new methods to ***QPDFAnnotationObjectHelper*** and ***QPDFFormFieldObjectHelper*** for querying flags and interpretation of different field types. Define constants in *qpdf/Constants.h* to help with interpretation of flag values.

  - Add new methods *QPDFAcroFormDocumentHelper::generateAppearancesIfNeeded* and *QPDFFormFieldObjectHelper::generateAppearance* for generating appearance streams. See discussion in *QPDFFormFieldObjectHelper.hh* for limitations.

  - Add two new helper functions for dealing with resource dictionaries: *QPDFObjectHandle::getResourceNames()* returns a list of all second-level keys, which correspond to the names of resources, and *QPDFOb-*

jectHandle::mergeResources() merges two resources dictionaries as long as they have non-conflicting keys. These methods are useful for certain types of objects that resolve resources from multiple places, such as form fields.

- Add methods *QPDFPageDocumentHelper::flattenAnnotations()* and *QPDFAnnotationObjectHelper::getPageContentForAppearance()* for handling low-level details of annotation flattening.

- Add new helper classes: ***QPDFOutlineDocumentHelper***, ***QPDFOutlineObjectHelper***, ***QPDFPageLabelDocumentHelper***, ***QPDFNameTreeObjectHelper***, and ***QPDFNumberTreeObjectHelper***.

- Add method *QPDFObjectHandle::getJSON()* that returns a JSON representation of the object. Call *serialize()* on the result to convert it to a string.

- Add a simple JSON serializer. This is not a complete or general-purpose JSON library. It allows assembly and serialization of JSON structures with some restrictions, which are described in the header file. This is the serializer used by qpdf's new JSON representation.

- Add new ***QPDFObjectHandle::Matrix*** class along with a few convenience methods for dealing with six-element numerical arrays as matrices.

- Add new method *QPDFObjectHandle::wrapInArray*, which returns the object itself if it is an array, or an array containing the object otherwise. This is a common construct in PDF. This method prevents you from having to explicitly test whether something is a single element or an array.

- Build Improvements

  - It is no longer necessary to run **autogen.sh** to build from a pristine checkout. Automatically generated files are now committed so that it is possible to build on platforms without autoconf directly from a clean checkout of the repository. The **configure** script detects if the files are out of date when it also determines that the tools are present to regenerate them.

  - Pull requests and the master branch are now built automatically in Azure Pipelines [https://dev.azure.com/qpdf/qpdf/_build], which is free for open source projects. The build includes Linux, mac, Windows 32-bit and 64-bit with mingw and MSVC, and an AppImage build. Official qpdf releases are now built with Azure Pipelines.

- Notes for Packagers

  - A new section has been added to the documentation with notes for packagers. Please see Section 2.4, “Notes for Packagers”, page 5.

  - The qpdf detects out-of-date automatically generated files. If your packaging system automatically refreshes libtool or autoconf files, it could cause this check to fail. To avoid this problem, pass **--disable-check-autofiles** to **configure**.

  - If you would like to have qpdf completion enabled automatically, you can install completion files in the distribution's default location. You can find sample completion files to install in the *completions* directory.

8.2.1: August 18, 2018

- Command-line Enhancements

  - Add **--keep-files-open=*[yn]*** to override default determination of whether to keep files open when merging. Please see the discussion of **--keep-files-open** in Section 3.3, “Basic Options”, page 7 for additional details.

8.2.0: August 16, 2018

- Command-line Enhancements
  - Add **--no-warn** option to suppress issuing warning messages. If there are any conditions that would have caused warnings to be issued, the exit status is still 3.
- Bug Fixes and Optimizations
  - Performance fix: optimize page merging operation to avoid unnecessary open/close calls on files being merged. This solves a dramatic slow-down that was observed when merging certain types of files.
  - Optimize how memory was used for the TIFF predictor, drastically improving performance and memory usage for files containing high-resolution images compressed with Flate using the TIFF predictor.
  - Bug fix: end of line characters were not properly handled inside strings in some cases.
  - Bug fix: using **--progress** on very small files could cause an infinite loop.
- API enhancements
  - Add new class ***QPDFSystemError***, derived from ***std::runtime_error***, which is now thrown by *QUtil::throw_system_error*. This enables the triggering ***errno*** value to be retrieved.
  - Add *ClosedFileInputSource::stayOpen* method, enabling a ***ClosedFileInputSource*** to stay open during manually indicated periods of high activity, thus reducing the overhead of frequent open/close operations.
- Build Changes
  - For the mingw builds, change the name of the DLL import library from *libqpdf.a* to *libqpdf.dll.a* to more accurately reflect that it is an import library rather than a static library. This potentially clears the way for supporting a static library in the future, though presently, the qpdf Windows build only builds the DLL and executables.

8.1.0: June 23, 2018

- Usability Improvements
  - When splitting files, qpdf detects fonts and images that the document metadata claims are referenced from a page but are not actually referenced and omits them from the output file. This change can cause a significant reduction in the size of split PDF files for files created by some software packages. In some cases, it can also make page splitting slower. Prior versions of qpdf would believe the document metadata and sometimes include all the images from all the other pages even though the pages were no longer present. In the unlikely event that the old behavior should be desired, or if you have a case where page splitting is very slow, the old behavior (and speed) can be enabled by specifying **--preserve-unreferenced-resources**. For additional details, please see Section 3.8, “Advanced Transformation Options”, page 17.
  - When merging multiple PDF files, qpdf no longer leaves all the files open. This makes it possible to merge numbers of files that may exceed the operating system's limit for the maximum number of open files.
  - The **--rotate** option's syntax has been extended to make the page range optional. If you specify **--rotate=*angle*** without specifying a page range, the rotation will be applied to all pages. This can be especially useful for adjusting a PDF created from a multi-page document that was scanned upside down.
  - When merging multiple files, the **--verbose** option now prints information about each file as it operates on that file.

    - When the **--progress** option is specified, qpdf will print a running indicator of its best guess at how far through the writing process it is. Note that, as with all progress meters, it's an approximation. This option is implemented in a way that makes it useful for software that uses the qpdf library; see API Enhancements below.

- Bug Fixes

    - Properly decrypt files that use revision 3 of the standard security handler but use 40 bit keys (even though revision 3 supports 128-bit keys).

    - Limit depth of nested data structures to prevent crashes from certain types of malformed (malicious) PDFs.

    - In "newline before endstream" mode, insert the required extra newline before the `endstream` at the end of object streams. This one case was previously omitted.

- API Enhancements

    - The first round of higher level "helper" interfaces has been introduced. These are designed to provide a more convenient way of interacting with certain document features than using ***QPDFObjectHandle*** directly. For details on helpers, see Section 7.3, "Helper Classes", page 34. Specific additional interfaces are described below.

    - Add two new document helper classes: ***QPDFPageDocumentHelper*** for working with pages, and ***QPDFAcroFormDocumentHelper*** for working with interactive forms. No old methods have been removed, but ***QPDFPageDocumentHelper*** is now the preferred way to perform operations on pages rather than calling the old methods in ***QPDFObjectHandle*** and ***QPDF*** directly. Comments in the header files direct you to the new interfaces. Please see the header files and *ChangeLog* for additional details.

    - Add three new object helper class: ***QPDFPageObjectHelper*** for pages, ***QPDFFormFieldObjectHelper*** for interactive form fields, and ***QPDFAnnotationObjectHelper*** for annotations. All three classes are fairly sparse at the moment, but they have some useful, basic functionality.

    - A new example program *examples/pdf-set-form-values.cc* has been added that illustrates use of the new document and object helpers.

    - The method *QPDFWriter::registerProgressReporter* has been added. This method allows you to register a function that is called by ***QPDFWriter*** to update your idea of the percentage it thinks it is through writing its output. Client programs can use this to implement reasonably accurate progress meters. The **qpdf** command line tool uses this to implement its **--progress** option.

    - New methods *QPDFObjectHandle::newUnicodeString* and *QPDFObject::unparseBinary* have been added to allow for more convenient creation of strings that are explicitly encoded using big-endian UTF-16. This is useful for creating strings that appear outside of content streams, such as labels, form fields, outlines, document metadata, etc.

    - A new class ***QPDFObjectHandle::Rectangle*** has been added to ease working with PDF rectangles, which are just arrays of four numeric values.

8.0.2: March 6, 2018

- When a loop is detected while following cross reference streams or tables, treat this as damage instead of silently ignoring the previous table. This prevents loss of otherwise recoverable data in some damaged files.

- Properly handle pages with no contents.