QPDF

Release 11.2.0

Jay Berkenbilt

Nov 21, 2022

CONTENTS:

1 What is QPDF? 3

2 License 5

3 Downloading QPDF 7

4 Building and Installing QPDF 9
- 4.1 Dependencies . . . 9
- 4.2 Build Instructions . . . 10
- 4.3 Build Options . . . 12
- 4.4 Crypto Providers . . . 15
- 4.5 Converting From autoconf to cmake . . . 17

5 Notes for Packagers 19
- 5.1 Build Options . . . 19
- 5.2 Package Tests . . . 19
- 5.3 Packaging Documentation . . . 20

6 Running qpdf 21
- 6.1 Basic Invocation . . . 21
- 6.2 Exit Status . . . 22
- 6.3 Shell Completion . . . 23
- 6.4 Help/Information . . . 23
- 6.5 General Options . . . 24
- 6.6 Advanced Control Options . . . 25
- 6.7 PDF Transformation . . . 27
- 6.8 Page Ranges . . . 31
- 6.9 PDF Modification . . . 32
- 6.10 Encryption . . . 35
- 6.11 Page Selection . . . 38
- 6.12 Overlay and Underlay . . . 41
- 6.13 Embedded Files/Attachments . . . 42
- 6.14 PDF Inspection . . . 44
- 6.15 JSON Options . . . 46
- 6.16 Options for Testing or Debugging . . . 48
- 6.17 Unicode Passwords . . . 49

7 QDF Mode 51

8 Using the QPDF Library 53
- 8.1 Using QPDF from C++ . . . 53

8.2 Using QPDF from other languages . . . 53
8.3 A Note About Unicode File Names . . . 54

9 Weak Cryptography **55**
9.1 Definition of Weak Cryptographic Algorithm . . . 55
9.2 Uses of Weak Encryption in qpdf . . . 56
9.3 Uses of Weak Hashing In QPDF . . . 56
9.4 API-Breaking Changes in qpdf 11.0 . . . 57

10 qpdf JSON **59**
10.1 Overview . . . 59
10.2 JSON Terminology . . . 60
10.3 What qpdf JSON is not . . . 60
10.4 qpdf JSON Format . . . 60
10.5 JSON Compatibility Guarantees . . . 67
10.6 JSON: Special Considerations . . . 68
10.7 Changes from JSON v1 to v2 . . . 69

11 Contributing to qpdf **71**
11.1 Source Repository . . . 71
11.2 Code Formatting . . . 71
11.3 Automated Tests . . . 72
11.4 Personal Comments . . . 73

12 Design and Library Notes **75**
12.1 Introduction . . . 75
12.2 Design Goals . . . 75
12.3 Helper Classes . . . 76
12.4 Implementation Notes . . . 77
12.5 QPDF Object Internals . . . 79
12.6 Casting Policy . . . 80
12.7 Encryption . . . 80
12.8 Random Number Generation . . . 81
12.9 Adding and Removing Pages . . . 81
12.10 Reserving Object Numbers . . . 82
12.11 Copying Objects From Other PDF Files . . . 82
12.12 Writing PDF Files . . . 82
12.13 Filtered Streams . . . 84
12.14 Object Accessor Methods . . . 84
12.15 Smart Pointers . . . 85

13 QPDFJob: a Job-Based Interface **91**
13.1 QPDFJob Design . . . 93

14 Linearization **95**
14.1 Basic Strategy for Linearization . . . 95
14.2 Preparing For Linearization . . . 95
14.3 Optimization . . . 95
14.4 Writing Linearized Files . . . 96
14.5 Calculating Linearization Data . . . 96
14.6 Known Issues with Linearization . . . 97
14.7 Debugging Note . . . 97

15 Object and Cross-Reference Streams **99**
15.1 Object Streams . . . 99

15.2 Cross-Reference Streams . . . 100
15.3 Implications for Linearized Files . . . 101
15.4 Implementation Notes . . . 101

16 PDF Encryption **103**
16.1 PDF Encryption Concepts . . . 103
16.2 PDF Encryption Details . . . 104
16.3 PDF Security Restrictions . . . 105
16.4 How qpdf handles security restrictions . . . 106
16.5 User and Owner Passwords . . . 107

17 Release Notes **109**

18 Acknowledgments **149**

19 Indices **151**

qpdf Command-line Options **153**

Welcome to the QPDF documentation! For the latest version of this documentation, please visit https://qpdf.readthedocs.io.

CHAPTER

ONE

WHAT IS QPDF?

QPDF is a program and C++ library for structural, content-preserving transformations on PDF files. QPDF's website is located at https://qpdf.sourceforge.io/. QPDF's source code is hosted on github at https://github.com/qpdf/qpdf. You can find the latest version of this documentation at https://qpdf.readthedocs.io/.

QPDF 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 of 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 that perhaps your original PDF creation tool 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.

This documentation aims to be comprehensive, but there is also a wiki for less polished material and ongoing work.

CHAPTER

TWO

LICENSE

QPDF is licensed under the Apache License, Version 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.

CHAPTER

THREE

DOWNLOADING QPDF

QPDF is included in most Linux distributions. Native packages are available for many other operating systems as well.

Other resources:

- GitHub release page
- GitHub project
- QPDF project web site

CHAPTER

FOUR

BUILDING AND INSTALLING QPDF

This chapter describes how to build and install qpdf.

4.1 Dependencies

qpdf has few external dependencies. This section describes what you need to build qpdf in various circumstances.

4.1.1 Basic Dependencies

- A C++ compiler that supports C++-17
- CMake version 3.16 or later
- zlib or a compatible zlib implementation
- A libjpeg-compatible library such as jpeg or libjpeg-turbo
- *Recommended but not required:* gnutls to be able to use the gnutls crypto provider and/or openssl to be able to use the openssl crypto provider

The qpdf source tree includes a few automatically generated files. The code generator uses Python 3. Automatic code generation is off by default. For a discussion, refer to *Build Options*.

4.1.2 Test Dependencies

qpdf's test suite is run by `ctest`, which is part of CMake, but the tests themselves are implemented using an embedded copy of qtest, which is implemented in perl. On Windows, MSYS2's perl is known to work.

qtest requires GNU diffutils or any other diff that supports **`diff -u`**. The default `diff` command works on GNU/Linux and MacOS.

Part of qpdf's test suite does comparisons of the contents PDF files by converting them to 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 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 setting the `QPDF_TEST_COMPARE_IMAGES` environment variable to 1 before running tests. Image comparison tests add these additional requirements:

- libtiff command-line utilities

- GhostScript version 8.60 or newer

Note: prior to qpdf 11, image comparison tests were enabled within `qpdf.test`, and you had to *disable* them by setting `QPDF_SKIP_TEST_COMPARE_IMAGES` to `1`. This was done automatically by `./configure`. Now you have to *enable* image comparison tests by setting an environment variable. This change was made because developers have to set the environment variable themselves now rather than setting it through the build. Either way, they are off by default.

4.1.3 Additional Requirements on Windows

- To build qpdf with Visual Studio, there are no additional requirements when the default cmake options are used. You can build qpdf from a Visual C++ command-line shell.
- To build with mingw, MSYS2 is recommended with the mingw32 and/or mingw64 tool chains. You can also build with MSVC from an MSYS2 environment.
- qpdf's test suite can run within the MSYS2 environment for both mingw and MSVC-based builds.

For additional notes, see `README-windows.md` in the source distribution.

4.1.4 Requirements for Building Documentation

The qpdf manual is written in reStructured Text and built with Sphinx using the Read the Docs Sphinx Theme. Versions of sphinx prior to version 4.3.2 probably won't work. Sphinx requires Python 3. In order to build the HTML documentation from source, you need to install sphinx and the theme, which you can typically do with `pip install sphinx sphinx_rtd_theme`. To build the PDF version of the documentation, you need `pdflatex`, `latexmk`, and a fairly complete LaTeX installation. Detailed requirements can be found in the Sphinx documentation. To see how the documentation is built for the qpdf distribution, refer to the `build-scripts/build-doc` file in the qpdf source distribution.

4.2 Build Instructions

Starting with qpdf 11, qpdf is built with CMake.

4.2.1 Basic Build Invocation

qpdf uses cmake in an ordinary way, so refer to the CMake documentation for details about how to run `cmake`. Here is a brief summary.

You can usually just run

```
cmake -S . -B build
cmake --build build
```

If you are using a multi-configuration generator such as MSVC, you should pass `--config <Config>` (where `<Config>` is `Release`, `Debug`, `RelWithDebInfo`, or `MinSizeRel` as discussed in the CMake documentation) to the *build* command. If you are running a single configuration generator such as the default Makefile generators in Linux or MSYS, you may want to pass `-DCMAKE_BUILD_TYPE=<Config>` to the original `cmake` command.

Run `ctest` to run the test suite. Since the real tests are implemented with qtest, you will want to pass `--verbose` to `cmake` so you can see the individual test outputs. Otherwise, you will see a small number of `ctest` commands that take a very long to run. If you want to run only a specific test file in a specific test suite, you can set the `TESTS` environment variable (used by `qtest-driver`) and pass the `-R` parameter to `ctest`. For example:

```
TESTS=qutil ctest --verbose -R libtests
```

would run only `qutil.test` from the `libtests` test suite.

4.2.2 Installation and Packaging

Installation can be performed using `cmake --install` or `cpack`. For most normal use cases, `cmake --install` or `cpack` can be run in the normal way as described in CMake documentation. qpdf follows all normal installation conventions and uses CMake-defined variables for standard behavior.

There are several components that can be installed separately:

Table 1: Installation Components

cli	Command-line tools
lib	The runtime libraries; required if you built with shared libraries
dev	Static libraries, header files, and other files needed by developers
doc	Documentation and, if selected for installation, the manual
examples	Example source files

Note that the `lib` component installs only runtime libraries, not header files or other files/links needed to build against qpdf. For that, you need `dev`. If you are using shared libraries, the `dev` will install files or create symbolic links that depend on files installed by `lib`, so you will need to install both. If you wanted to build software against the qpdf library and only wanted to install the files you needed for that purpose, here are some examples:

- Install development files with static libraries only:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=RelWithDebInfo -DBUILD_SHARED_LIBS=OFF
cmake --build build --parallel --target libqpdf
cmake --install build --component dev
```

- Install development files with shared libraries only:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=RelWithDebInfo -DBUILD_STATIC_LIBS=OFF
cmake --build build --parallel --target libqpdf
cmake --install build --component lib
cmake --install build --component dev
```

- Install development files with shared and static libraries:

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=RelWithDebInfo
cmake --build build --parallel --target libqpdf libqpdf_static
cmake --install build --component lib
cmake --install build --component dev
```

There are also separate options, discussed in *Build Options*, that control how certain specific parts of the software are installed.

4.3 Build Options

All available build options are defined in the the top-level `CMakeLists.txt` file and have help text. You can see them using any standard cmake front-end (like `cmake-gui` or `ccmake`). This section describes options that apply to most users. If you are trying to map autoconf options (from prior to qpdf 11) to cmake options, please see *Converting From autoconf to cmake*.

If you are packaging qpdf for a distribution, you should also read *Notes for Packagers*.

4.3.1 Basic Build Options

BUILD_DOC
Whether to build documentation with sphinx. You must have the required tools installed.

BUILD_DOC_HTML
Visible when BUILD_DOC is selected. This option controls building HTML documentation separately from PDF documentation since the sphinx theme is only needed for the HTML documentation.

BUILD_DOC_PDF
Visible when BUILD_DOC is selected. This option controls building PDF documentation separately from HTML documentation since additional tools are required to build the PDF documentation.

BUILD_SHARED_LIBS, BUILD_STATIC_LIBS
You can configure whether to build shared libraries, static libraries, or both. You must select at least one of these options. For rapid iteration, select only one as this cuts the build time in half.

On Windows, if you build with shared libraries, you must have the output directory for libqpdf (e.g. `libqpdf/Release` or `libqpdf` within the build directory) in your path so that the compiled executables can find the DLL. Updating your path is not necessary if you build with static libraries only.

QTEST_COLOR
Turn this on or off to control whether qtest uses color in its output.

4.3.2 Options for Working on qpdf

CHECK_SIZES
The source file `qpdf/sizes.cc` is used to display the sizes of all objects in the public API. Consistency of its output between releases is used as part of the check against accidental breakage of the binary interface (ABI). Turning this on causes a test to be run that ensures an exact match between classes in `sizes.cc` and classes in the library's public API. This option requires Python 3.

ENABLE_QTC
This is off by default, except in maintainer mode. When off, `QTC::TC` calls are compiled out by having `QTC::TC` be an empty inline function. The underlying `QTC::TC` remains in the library, so it is possible to build and package the qpdf library with `ENABLE_QTC` turned off while still allowing developer code to use `QTC::TC` if desired. If you are modifying qpdf code, it's a good idea to have this on for more robust automated testing. Otherwise, there's no reason to have it on.

GENERATE_AUTO_JOB
Some qpdf source files are automatically generated from `job.yml` and the CLI documentation. If you are adding new command-line arguments to the qpdf CLI or updating `manual/cli.rst` in the qpdf sources, you should turn this on. This option requires Python 3.

WERROR
Make any compiler warnings into errors. We want qpdf to compile free of warnings whenever possible, but

there's always a chance that a compiler upgrade or tool change may cause warnings to appear that weren't there before. If you are testing qpdf with a new compiler, you should turn this on.

4.3.3 Environment-Specific Options

SHOW_FAILED_TEST_OUTPUT
Ordinarily, qtest (which drives qpdf's test suite) writes detailed information about its output to the file `qtest.log` in the build output directory. If you are running a build in a continuous integration or automated environment where you can't get to those files, you should enable this option and also run `ctest --verbose` or `ctest --output-on-failure`. This will cause detailed test failure output to be written into the build log.

CI_MODE
Turning this on sets options used in qpdf's continuous integration environment to ensure we catch as many problems as possible. Specifically, this option enables `SHOW_FAILED_TEST_OUTPUT` and `WERROR` and forces the native crypto provider to be built.

MAINTAINER_MODE
Turning this option on sets options that should be on if you are maintaining qpdf. In turns on the following:

- `BUILD_DOC`
- `CHECK_SIZES`
- `ENABLE_QTC`
- `GENERATE_AUTO_JOB`
- `WERROR`
- `REQUIRE_NATIVE_CRYPTO`

It is possible to turn `BUILD_DOC` off in maintainer mode so that the extra requirements for building documentation don't have to be available.

4.3.4 Build-time Crypto Selection

Since version 9.1.0, qpdf can use external crypto providers in addition to its native provider. For a general discussion, see *Crypto Providers*. This section discusses how to configure which crypto providers are compiled into qpdf.

In nearly all cases, external crypto providers should be preferred over the native one. However, if you are not concerned about working with encrypted files and want to reduce the number of dependencies, the native crypto provider is fully supported.

By default, qpdf's build enables every external crypto providers whose dependencies are available and only enables the native crypto provider if no external providers are available. You can change this behavior with the options described here.

USE_IMPLICIT_CRYPTO
This is on by default. If turned off, only explicitly selected crypto providers will be built. You must use at least one of the `REQUIRE` options below.

ALLOW_CRYPTO_NATIVE
This option is only available when USE_IMPLICIT_CRYPTO is selected, in which case it is on by default. Turning it off prevents qpdf from falling back to the native crypto provider when no external provider is available.

REQUIRE_CRYPTO_NATIVE
Build the native crypto provider even if other options are available.

REQUIRE_CRYPTO_GNUTLS
Require the gnutls crypto provider. Turning this on makes in an error if the gnutls library is not available.

REQUIRE_CRYPTO_OPENSSL
Require the openssl crypto provider. Turning this on makes in an error if the openssl library is not available.

DEFAULT_CRYPTO
Explicitly select which crypto provider is used by default. See *Runtime Crypto Provider Selection* for information about run-time selection of the crypto provider. If not specified, qpdf will pick gnutls if available, otherwise openssl if available, and finally native as a last priority.

Example: if you wanted to build with only the gnutls crypto provider, you should run cmake with `-DUSE_IMPLICIT_CRYPTO=0 -DREQUIRE_CRYPTO_GNUTLS=1`.

4.3.5 Advanced Build Options

These options are used only for special purposes and are not relevant to most users.

AVOID_WINDOWS_HANDLE
Disable use of the `HANDLE` type in Windows. This can be useful if you are building for certain embedded Windows environments. Some functionality won't work, but you can still process PDF files from memory in this configuration.

BUILD_DOC_DIST, INSTALL_MANUAL
By default, installing qpdf does not include a pre-built copy of the manual. Instead, it installs a README file that tells people where to find the manual online. If you want to install the manual, you must enable the `INSTALL_MANUAL` option, and you must have a `doc-dist` directory in the manual directory of the build. The `doc-dist` directory is created if `BUILD_DOC_DIST` is selected and `BUILD_DOC_PDF` and `BUILD_DOC_HTML` are both on.

The `BUILD_DOC_DIST` and `INSTALL_MANUAL` options are separate and independent because of the additional tools required to build documentation. In particular, for qpdf's official release preparation, a `doc-dist` directory is built in Linux and then extracted into the Windows builds so that it can be included in the Windows installers. This prevents us from having to build the documentation in a Windows environment. For additional discussion, see *Documentation Packaging Rationale*.

INSTALL_CMAKE_PACKAGE
Controls whether or not to install qpdf's cmake configuration file (on by default).

INSTALL_EXAMPLES
Controls whether or not to install qpdf's example source files with documentation (on by default).

INSTALL_PKGCONFIG
Controls whether or not to install qpdf's pkg-config configuration file (on by default).

OSS_FUZZ
Turning this option on changes the build of the fuzzers in a manner specifically required by Google's oss-fuzz project. There is no reason to turn this on for any other reason. It is enabled by the build script that builds qpdf from that context.

SKIP_OS_SECURE_RANDOM, USE_INSECURE_RANDOM
The native crypto implementation uses the operating systems's secure random number source when available. It is not used when an external crypto provider is in use. If you are building in a very specialized environment where you are not using an external crypto provider but can't use the OS-provided secure random number generator, you can turn both of these options on. This will cause qpdf to fall back to an insecure random number generator, which may generate guessable random numbers. The resulting qpdf is still secure, but encrypted files may be more subject to brute force attacks. Unless you know you need these options for a specialized purpose, you don't need them. These options were added to qpdf in response to a special request from a user who needed to run a specialized PDF-related task in an embedded environment that didn't have a secure random number source.

4.3.6 Building without wchar_t

It is possible to build qpdf on a system that doesn't have `wchar_t`. The resulting build of qpdf is not API-compatible with a regular qpdf build, so this option cannot be selected from cmake. This option was added to qpdf to support installation on a very stripped down embedded environment that included only a partial implementation of the standard C++ library.

You can disable use of `wchar_t` in qpdf's code by defining the `QPDF_NO_WCHAR_T` preprocessor symbol in your build (e.g. by including `-DQPDF_NO_WCHAR_T` in `CFLAGS` and `CXXFLAGS`).

While `wchar_t` is part of the C++ standard library and should be present on virtually every system, there are some stripped down systems, such as those targeting certain embedded environments, that lack `wchar_t`. Internally, qpdf uses UTF-8 encoding for everything, so there is nothing important in qpdf's API that uses `wchar_t`. However, there are some helper methods for converting between `wchar_t*` and `char*`.

If you are building in an environment that does not support `wchar_t`, you can define the preprocessor symbol `QPDF_NO_WCHAR_T` in your build. This will work whether you are building qpdf and need to avoid compiling the code that uses wchar_t or whether you are building client code that uses qpdf.

Note that, when you build code with libqpdf, it is *not necessary* to have the definition of `QPDF_NO_WCHAR_T` in your build match what was defined when the library was built as long as you are not calling any of the methods that use `wchar_t`.

4.4 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. Some of these are weak cryptographic algorithms. For a discussion of why they're needed, see *Weak Cryptography*.

The available crypto provider implementations are `gnutls`, `openssl`, and `native`. OpenSSL support was added in qpdf 10.0.0 with support for OpenSSL added in 10.4.0. GnuTLS support was introduced in qpdf 9.1.0. Additional implementations can be added as needed. It is also possible for a developer to provide their own implementation without modifying the qpdf library.

For information about selecting which crypto providers are compiled into qpdf, see *Build-time Crypto Selection*.

4.4.1 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.

4.4.2 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`.

4.4.3 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 nondeterministic 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.

4.5 Converting From autoconf to cmake

Versions of qpdf before qpdf 11 were built with `autoconf` and a home-grown GNU Make-based build system. If you built qpdf with special `./configure` options, this section can help you switch them over to `cmake`.

In most cases, there is a one-to-one mapping between configure options and cmake options. There are a few exceptions:

- The cmake build behaves differently with respect to whether or not to include support for the native crypto provider. Specifically, it is not implicitly enabled unless explicitly requested if there are other options available. You can force it to be included by enabling `REQUIRE_CRYPTO_NATIVE`. For details, see *Build-time Crypto Selection*.
- The `--enable-external-libs` option is no longer available. The cmake build detects the presence of `external-libs` automatically. See `README-windows.md` in the source distribution for a more in-depth discussion.
- The sense of the option representing use of the OS-provided secure random number generator has been reversed: the `--enable-os-secure-random`, which was on by default, has been replaced by the `SKIP_OS_SECURE_RANDOM` option, which is off by default. The option's new name and behavior match the preprocessor symbol that it turns on.
- Non-default test configuration is selected with environment variables rather than cmake. The old `./configure` options just set environment variables. Note that the sense of the variable for image comparison tests has been reversed. It used to be that you had to set `QPDF_SKIP_TEST_COMPARE_IMAGES` to 1 to *disable* image comparison tests. This was done by default. Now you have to set `QPDF_TEST_COMPARE_IMAGES` to 1 to *enable* image comparison tests. Either way, they are off by default.
- Non-user-visible change: the preprocessor symbol that triggers the export of functions into the public ABI (application binary interface) has been changed from `DLL_EXPORT` to `libqpdf_EXPORTS`. This detail is encapsulated in the build and is only relevant to people who are building qpdf on their own or who may have previously needed to work around a collision between qpdf's use of `DLL_EXPORT` and someone else's use of the same symbol.
- A handful of options that were specific to autoconf or the old build system have been dropped.
- `cmake --install` installs example source code in `doc/qpdf/examples` in the `examples` installation component. Packagers are encouraged to package this with development files if there is no separate doc package. This can be turned off by disabling the `INSTALL_EXAMPLES` build option.

There are some new options available in the cmake build that were not available in the autoconf build. This table shows the old options and their equivalents in cmake.

Table 2: configure flags to cmake options

enable-avoid-windows-handle	AVOID_WINDOWS_HANDLE
enable-check-autofiles	none – not relevant to cmake
enable-crypto-gnutls	REQUIRE_CRYPTO_GNUTLS
enable-crypto-native	REQUIRE_CRYPTO_NATIVE (but see above)
enable-crypto-openssl	REQUIRE_CRYPTO_OPENSSL
enable-doc-maintenance	BUILD_DOC
enable-external-libs	none – detected automatically
enable-html-doc	BUILD_DOC_HTML
enable-implicit-crypto	USE_IMPLICIT_CRYPTO
enable-insecure-random	USE_INSECURE_RANDOM
enable-ld-version-script	none – detected automatically
enable-maintainer-mode	MAINTAINER_MODE (slight differences)
enable-os-secure-random (on by default)	SKIP_OS_SECURE_RANDOM (off by default)
enable-oss-fuzz	OSS_FUZZ
enable-pdf-doc	BUILD_DOC_PDF
enable-rpath	none – cmake handles rpath correctly
enable-show-failed-test-output	SHOW_FAILED_TEST_OUTPUT
enable-test-compare-images	set the `QPDF_TEST_COMPARE_IMAGES` environment variable
enable-werror	WERROR
with-buildrules	none – not relevant to cmake
with-default-crypto	DEFAULT_CRYPTO
large-file-test-path	set the `QPDF_LARGE_FILE_TEST_PATH` environment variable

CHAPTER

FIVE

NOTES FOR PACKAGERS

If you are packaging qpdf for an operating system distribution, this chapter is for you. Otherwise, feel free to skip.

5.1 Build Options

For a detailed discussion of build options, please refer to *Build Options*. This section calls attention to options that are particularly useful to packagers.

- Perl must be present at build time. Prior to qpdf version 9.1.1, there was a runtime dependency on perl, but this is no longer the case.
- Make sure you are getting the intended behavior with regard to crypto providers. Read *Build-time Crypto Selection* for details.
- Use of `SHOW_FAILED_TEST_OUTPUT` is recommended for building in continuous integration or other automated environments as it makes it possible to see test failures in build logs. This should be combined with either `ctest --verbose` or `ctest --output-on-failure`.
- qpdf's install targets do not install completion files by default since there is no standard location for them. 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. See the `completions/README.md` file for more information.
- Starting with qpdf 11, qpdf's default installation installs source files from the examples directory with documentation. Prior to qpdf 11, this was a recommendation for packagers but was not done automatically.

5.2 Package Tests

The `pkg-test` directory contains very small test shell scripts that are designed to help smoke-test an installation of qpdf. They were designed to be used with debian's autopkgtest framework but can be used by others. Please see `pkg-test/README.md` in the source distribution for details.

5.3 Packaging Documentation

Starting in qpdf version 10.5, pre-built documentation is no longer distributed with the qpdf source distribution. Here are a few options you may want to consider for your packages:

- **Do nothing**

  When you run `make install`, the file `README-doc.txt` is installed in the documentation directory. That file tells the reader where to find the documentation online and where to go to download offline copies of the documentation. This is the option selected by the debian packages.

- **Embed pre-built documentation**

  You can obtain pre-built documentation and extract its contents into your distribution. This is what the Windows binary distributions available from the qpdf release site do. You can find the pre-built documentation in the release area in the file `qpdf-`*`version`*`-doc.zip`. For an example of this approach, look at qpdf's GitHub actions build scripts. The `build-scripts/build-doc` script builds with `-DBUILD_DOC_DIST=1` to create the documentation distribution. The `build-scripts/build-windows` script extracts it into the build tree and builds with `-DINSTALL_MANUAL=1` to include it in the installer.

- **Build the documentation yourself**

  You can build the documentation as part of your build process. Be sure to pass `-DBUILD_DOC_DIST=1` and `-DINSTALL_MANUAL=1` to cmake. This is what the AppImage build does. The latest version of Sphinx at the time of the initial conversion a sphinx-based documentation was 4.3.2. Older versions are not guaranteed to work.

5.3.1 Documentation Packaging Rationale

This section describes the reason for things being the way they are. It's for information only; you don't have to know any of this to package qpdf.

What is the reason for this change? Prior to qpdf 10.5, the qpdf manual was a docbook XML file. The generated documents were the product of running the file through build-time style sheets and contained no copyrighted material of their own. Starting with version 10.5, the manual is written in reStructured Text and built with Sphinx. This change was made to make it much easier to automatically generate portions of the documentation and to make the documentation easier to work with. The HTML output of Sphinx is also much more readable, usable, and suitable for online consumption than the output of the docbook style sheets. The downsides are that the generated HTML documentation now contains Javascript code and embedded fonts, and the PDF version of the documentation is no longer as suitable for printing (at least as of the 10.5 distribution) since external link targets are no longer shown and cross references no longer contain page number information. The presence of copyrighted material in the generated documentation, even though things are licensed with MIT and BSD licenses, complicates the job of the packager in various ways. For one thing, it means the `NOTICE.md` file in the source repository would have to keep up with the copyright information for files that are not controlled in the repository. Additionally, some distributions (notably Debian/Ubuntu) discourage inclusion of sphinx-generated documentation in packages, preferring you instead to build the documentation as part of the package build process and to depend at runtime on a shared package that contains the code. At the time of the conversion of the qpdf manual from docbook to sphinx, newer versions of both sphinx and the html theme were required than were available in some of most of the Debian/Ubuntu versions for which qpdf was packaged.

Since always-on Internet connectivity is much more common than it used to be, many users of qpdf would prefer to consume the documentation online anyway, and the lack of pre-built documentation in the distribution won't be as big of a deal. However there are still some people who can't or choose not to view documentation online. For them, pre-built documentation is still available.

CHAPTER

SIX

RUNNING QPDF

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

6.1 Basic Invocation

```
Usage: qpdf [infile] [options] [outfile]
```

The **qpdf** command reads the PDF file *infile*, applies various transformations or modifications to the file in memory, and writes the result to *outfile*. When run with no options, the output file is functionally identical to the input file but may be structurally reorganized, and orphaned objects are removed from the file. Many options are available for applying transformations or modifications to the file.

infile can be a regular file, or it can be *--empty* to start with an empty PDF file. There is no way to use standard input since the input file has to be seekable.

outfile can be a regular file, - to represent standard output, or *--replace-input* to indicate that the input file should be overwritten. The output file does not have to be seekable, even when generating linearized files. You can also use *--split-pages* to create separate output files for each page (or group of pages) instead of a single output file.

Password-protected files may be opened by specifying a password with *--password*.

All options other than help options (see *Help/Information*) require an input file. If inspection or JSON options (see *PDF Inspection* and *JSON Options*) or help options are given, an output file must not be given. Otherwise, an output file is required.

If `@filename` appears as a word anywhere in the command-line, it will be read line by line, and each line will be treated as a command-line argument. Leading and trailing whitespace is intentionally not removed from lines, which makes it possible to handle arguments that start or end with spaces. 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, though see also *--password-file*. Note that the `@filename` can't appear in the middle of an argument, so constructs such as `--arg=@filename` will not work. Instead, you would have to include the option and its parameter (e.g., `--option=parameter`) as a line in the `filename` file and just pass `@filename` on the command line.

6.1.1 Related Options

--empty

This option may be given in place of *infile*. This causes qpdf to use a dummy input file that contains zero pages. This option is useful in conjunction with *--pages*. See *Page Selection* for details.

--replace-input

This option may be given in place of *outfile*. This causes 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. If there are errors, the input file is left untouched.

--job-json-file=file

Specify the name of a file whose contents are expected to contain a QPDFJob JSON file. This file is read and treated as if the equivalent command-line arguments were supplied. It can be repeated and mixed freely with other options. Run qpdf with *--job-json-help* for a description of the job JSON input file format. For more information, see *QPDFJob: a Job-Based Interface*. Note that this is unrelated to *--json* but may be combined with it.

6.2 Exit Status

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

Table 1: Exit Codes

0	no errors or warnings were found
1	not used
2	errors were found; the file was not processed
3	warnings were found without errors

Notes:

- A PDF file may have problems that qpdf can't detect.
- With the *--warning-exit-0* option, exit status 0 is used even if there are warnings.
- **qpdf** does not exit with status 1 since the shell uses this exit code if it is unable to invoke the command.
- If both errors and warnings were found, exit status 2 is used.
- The *--is-encrypted* and *--requires-password* options use different exit codes. See their help for details.

6.2.1 Related Options

--warning-exit-0

If there were warnings only and no errors, exit with exit code 0 instead of 3. When combined with *--no-warn*, the effect is for **qpdf** to completely ignore warnings.

6.3 Shell Completion

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 use an absolute path to qpdf in the above invocation. 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 a copy of qpdf that is run directly out of the source tree or that is invoked with a wrapper script. You can specify a full path to the qpdf you want to use for completion in the `QPDF_EXECUTABLE` environment variable.

6.3.1 Related Options

--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.

6.4 Help/Information

Help options provide some information about qpdf itself. Help options are only valid as the first and only command-line argument.

6.4.1 Related Options

--help[=--option|topic]

Display command-line invocation help. Use `--help=`*`--option`* for help on a specific option and `--help=`*`topic`* for help on a help topic and also provides a list of available help topics.

--version

Display the version of qpdf. The version number displayed is the one that is compiled into the qpdf library. If you don't see the version number you expect, you may have more than one version of **qpdf** installed and may not have your library path set up correctly.

--copyright

Display copyright and license information.

--show-crypto

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

--job-json-help

Describe the format of the QPDFJob JSON input used by *`--job-json-file`*. For more information about QPDFJob, see *QPDFJob: a Job-Based Interface*.

6.5 General Options

This section describes general options that control **qpdf**'s behavior. They are not necessarily related to the specific operation that is being performed and may be used whether or not an output file is being created.

6.5.1 Related Options

--password=password

Specifies a password for accessing encrypted, password-protected files. To read the password from a file or standard input, you can use *--password-file*.

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 *Unicode Passwords*. Previous versions of this manual described workarounds using the **iconv** command. Such workarounds are no longer required or recommended starting with qpdf 8.4.0. However, for backward compatibility, qpdf attempts to detect those workarounds and do the right thing in most cases.

--password-file=filename

Reads the first line from the specified file and uses it as the password for accessing encrypted files. *filename* may be - to read the password from standard input, but if you do that the password is echoed and there is no prompt, so use - with caution. Note that leading and trailing spaces are not stripped from the password.

--verbose

Increase verbosity of output. This includes information about files created, image optimization, and several other operations. In some cases, it also displays additional information when inspection options (see *PDF Inspection*) are used.

--progress

Indicate progress while writing output files. Progress indication does not start until writing starts, so there may be a delay before progress indicators are seen if complicated transformations are being applied before the write process begins.

--no-warn

Suppress writing of warnings to stderr. If warnings were detected and suppressed, **qpdf** will still exit with exit code 3. To completely ignore warnings, also specify *--warning-exit-0*. Use with caution as qpdf is not always successful in recovering from situations that cause warnings to be issued.

--deterministic-id

Generate a secure, random document ID using deterministic values. 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. This option can be useful for testing. See also *--static-id*.

While qpdf will generate the same deterministic ID given the same output PDF, there is no guarantee that different versions of qpdf will generate exactly the same PDF output for the same input and options. While care is taken to avoid gratuitous changes to qpdf's PDF generation, new versions of qpdf may include changes or bug fixes that cause slightly different PDF code to be generated. Such changes are noted in the release notes.

--allow-weak-crypto

Encrypted PDF files using 40-bit keys or 128-bit keys without AES use the insecure *RC4* encryption algorithm. Starting with version 11.0, qpdf's default behavior is to refuse to write files using RC4 encryption. Use this

option to allow creation of such files. In versions 10.4 through 10.6, attempting to create weak encrypted files was a warning, rather than an error, without this flag. See *Weak Cryptography* for additional details.

No check is performed for weak crypto when preserving encryption parameters from or copying encryption parameters from other files. The rationale for this is discussed in *Weak Cryptography*.

--keep-files-open=[y|n]

This option controls whether qpdf keeps individual files open while merging. By default, qpdf keeps files open when merging unless more than 200 files are specified, in which case files are opened as needed and closed when finished. Repeatedly opening and closing files may impose a large performance penalty with some file systems, especially networked file systems. If you know that you have a large enough open file limit and are suffering from performance problems, or if you have an open file limit smaller than 200, you can use this option to override the default behavior by specifying `--keep-files-open=y` to force **qpdf** to keep files open or `--keep-files-open=n` to force it to only open files as needed. See also *--keep-files-open-threshold*.

Historical note: 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. The current behavior was introduced in qpdf version 8.2.1.

--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.

6.6 Advanced Control Options

Advanced control options control qpdf's behavior in ways that would normally never be needed by a user but that may be useful to developers or people investigating problems with specific files.

6.6.1 Related Options

--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 parameter 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 encoded with the wrong character encoding. This option suppresses that behavior. Under normal conditions, there are no reasons to use this option. See *Unicode Passwords* 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 *Unicode Passwords*. 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 of 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 a `@filename` option or *`--password-file`* to do it, in which case this option forces qpdf to respect the string of bytes as provided. Note that this option may cause 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.

`--suppress-recovery`

Prevents qpdf from attempting to reconstruct a file's cross reference table when there are errors reading objects from the file. Recovery is triggered by a variety of situations. While usually successful, it uses heuristics that don't work on all files. If this option is given, **`qpdf`** fails on the first error it encounters.

`--ignore-xref-streams`

Tells qpdf to ignore any cross-reference streams, falling back to any embedded cross-reference tables or triggering document recovery. Ordinarily, qpdf reads cross-reference streams when they are present in a PDF file. If this option 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.

6.7 PDF Transformation

The options discussed in this section tell qpdf to apply transformations that change the structure of a PDF file without changing its content. Examples include creating linearized (web-optimized) files, adding or removing encryption, restructuring files for older viewers, and rewriting files for human inspection. See also *PDF Modification*.

6.7.1 Related Options

--linearize

Create linearized (web-optimized) output files. Linearized files are formatted in a way that allows compliant readers to begin displaying a PDF file before it is fully downloaded. Ordinarily, the entire file must be present before it can be rendered because important cross-reference information typically appears at the end of the file.

--encrypt user-password owner-password key-length [options] --

This flag starts encryption options, used to create encrypted files. Please see *Encryption* for details.

--decrypt

Create an output file with no encryption even if the input file is encrypted. This option overrides the default behavior of preserving whatever encryption was present on the input file. This functionality is not intended to be used for bypassing copyright restrictions or other restrictions placed on files by their producers. See also *--copy-encryption*.

--copy-encryption=file

Copy all encryption parameters, including the user password, the owner password, and all security restrictions, from the specified file instead of preserving the encryption details from the input file. This works even if only one of the user password or owner password is known. If the encryption file requires a password, use the *--encryption-file-password* option to set it. 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. This option can be useful if you need to decrypt a file to make manual changes to it or to change it outside of qpdf, and then want to restore the original encryption on the file without having to manual specify all the individual settings. See also *--decrypt*.

Checks for weak cryptographic algorithms are intentionally not made by this operation. See *Weak Cryptography* for the rationale.

--encryption-file-password=password

If the file specified with *--copy-encryption* requires a password, supply the password using this option. This option is necessary because the *--password* option applies to the input file, not the file from which encryption is being copied.

--qdf

Create a PDF file suitable for viewing and editing in a text editor. This is to edit the PDF code, not the page contents. To edit a QDF file, your text editor must preserve binary data. In a QDF file, all streams that can be uncompressed are uncompressed, and content streams are normalized, among other changes. The companion tool **fix-qdf** can be used to repair hand-edited QDF files. QDF is a feature specific to the qpdf tool. For additional information, see *QDF Mode*. Note that *--linearize* disables QDF mode.

QDF mode has full support for object streams, but sometimes it's easier to locate a specific object if object streams are disabled. When trying to understand some PDF construct by inspecting an existing file, it can be useful to combine `--qdf` with `--object-streams=disable`.

This flag changes some of the defaults of other options: stream data is uncompressed, content streams are normalized, and encryption is removed. These defaults can still be overridden by specifying the appropriate options with `--qdf`. Additionally, in QDF mode, stream lengths are stored as indirect objects, objects are formatted 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. When editing QDF files, it is not necessary to maintain the object formatting.

When normalizing content, if qpdf runs into any lexical errors, it will print a warning indicating that content may be damaged. If you want to create QDF files without content normalization, specify `--qdf --normalize-content=n`. You can also create a non-QDF file with uncompressed streams using `--stream-data=uncompress`. Either option will 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.

See also *--no-original-object-ids*.

--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. The original object ID comment is there by default because it makes it easier to trace objects back to the original file.

--compress-streams=[y|n]

By default, or with `--compress-streams=y`, qpdf will compress streams using the flate compression algorithm (used by zip and gzip) unless those streams are compressed in some other way. This analysis is made after qpdf attempts to uncompress streams and is therefore closely related to *--decode-level*. To suppress this behavior and leave streams streams uncompressed, use `--compress-streams=n`. In QDF mode (see *QDF Mode* and *--qdf*), the default is to leave streams uncompressed.

--decode-level=parameter

Controls which streams qpdf tries to decode. The default is `generalized`.

The following values for *parameter* are available:

- `none`: do not attempt to decode any streams. This is the default with *--json-output*.
- `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. This is the default except when *--json-output* is given.
- `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)

There are several filters that **qpdf** does not support. These are left untouched regardless of the option. Future versions of qpdf may support additional filters.

Because the default value is `generalized`, qpdf's default behavior is to uncompress any stream that is encoded using non-lossy filters that qpdf understands. If `--compress-streams=y` is also in effect, which is the default (see *--compress-streams*), the overall effect is that qpdf will recompress streams with generalized filters using flate compression, effectively eliminating LZW and ASCII-based filters. This is usually desirable behavior but can be disabled with `--decode-level=none`. Note that `--decode-level=none` is the default when *--json-output* is specified, but it can be overridden in that case as well.

As a special case, streams already compressed with `/FlateDecode` are not uncompressed and recompressed. You can change this behavior with *--recompress-flate*.

--stream-data=parameter

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

The default generalized compression scheme used by PDF is flate (`/FlateDecode`), which is the same as used by **zip** and **gzip**. Usually qpdf just leaves these alone. This option tells **qpdf** to uncompress and recompress streams compressed with flate. This can be useful when combined with *--compression-level*. Using this option may make **qpdf** much slower when writing output files.

--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. Lower numbers compress less and are faster; higher numbers compress more and are slower. Note that **qpdf** doesn't uncompress and recompress streams compressed with flate 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`. If you omit this option, qpdf defers to the compression library's default behavior.

--normalize-content=[y|n]

Enables or disables normalization of newlines in PDF content streams to UNIX-style newlines, which is useful for viewing files in a programmer-friendly text edit across multiple platforms. Content normalization is off by default, but is automatically enabled by *--qdf* (see also *QDF Mode*). It is not recommended to use this option for production use. If qpdf runs into any lexical errors while normalizing content, it will print a warning indicating that content may be damaged.

--object-streams=mode

Controls handling of object streams. The value of *mode* may be one of the following:

Table 2: Object Stream Modes

`preserve`	preserve original object streams, if any (the default)
`disable`	create output files with no object streams
`generate`	create object streams, and compress objects when possible

Object streams are PDF streams that contain other objects. Putting objects into object streams allows the PDF objects themselves to be compressed, which can result in much smaller PDF files. Combining this option with *--compression-level* and *--recompress-flate* can often result in the creation of smaller PDF files.

Object streams, also known as compressed objects, were introduced into the PDF specification at version 1.5 around 2003. Some ancient PDF viewers may not support files with object streams. qpdf can be used to transform files with object streams into files without object streams or vice versa.

In `preserve` mode, the relationship between objects and the streams that contain them is preserved from the original file. If the file has no object streams, qpdf will not add any. In `disable` mode, all objects are written as regular, uncompressed objects. The resulting file should be structurally readable by older PDF viewers, though there is still a chance that the file may contain other content that some older readers can't support. In `generate` mode, qpdf will create its own object streams. This will usually result in more compact PDF files. In this mode, qpdf will also make sure the PDF version number in the header is at least 1.5.

--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. Disabling this default behavior 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 ordered 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=parameter

Parameters: `auto` (the default), `yes`, or `no`.

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 qpdf 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 such a case, please report it as a bug at https://github.com/qpdf/qpdf/issues/.

--preserve-unreferenced-resources

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

See also *--preserve-unreferenced*, which does something completely different. To reduce confusion, you should use `--remove-unreferenced-resources=no` instead.

--newline-before-endstream

Tell 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.

--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 be combined with QDF mode or content normalization to make it easier to look at all of a page's contents at once. It is common for PDF writers to create multiple content streams for a variety of reasons such as making it easier to modify page contents and splitting very large content streams so PDF viewers may be able to use less memory.

--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 unless *--keep-inline-images* is also specified.

--ii-min-bytes=size-in-bytes

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

--min-version=version

Force 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*]. If *.extension-level*, is given, 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. **qpdf** does not check whether the specified version is actually required.

--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 the deprecated, unsupported "R5" format is allowed with extension levels 3 through 7), 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 they don't support. 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.

6.8 Page Ranges

Several **qpdf** command-line options use page ranges. This section describes the syntax of a page range.

- A plain number indicates a page numbered from 1, so `1` represents the first page.
- A number preceded by `r` counts from the end, so `r1` is the last page, `r2` is the second-to-last page, etc.
- The letter `z` represents the last page and is the same as `r1`.
- Page numbers may appear in any order separated by commas.
- Two page numbers separated by dashes represents the inclusive range of pages from the first to the second. If the first number is higher than the second number, it is the range of pages in reverse.
- The range may be appended with `:odd` or `:even` to select only pages from the resulting range in odd or even positions. In this case, odd and even refer to positions in the final range, not whether the original page number is odd or even.

Table 3: Example Page Ranges

`1,6,4`	pages 1, 6, and 4 in that order
`3-7`	pages 3 through 7 inclusive in increasing order
`7-3`	pages 7, 6, 5, 4, and 3 in that order
`1-z`	all pages in order
`z-1`	all pages in reverse order
`1,3,5-9,15-12`	pages 1, 3, 5, 6, 7, 8, 9, 15, 14, 13, and 12 in that order
`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`	pages 5, 7, 8, 9, and 12
`5,7-9,12:odd`	pages 5, 8, and 12, which are the pages in odd positions from the original set of 5, 7, 8, 9, 12
`5,7-9,12:even`	pages 7 and 9, which are the pages in even positions from the original set of 5, 7, 8, 9, 12

6.9 PDF Modification

Modification options make systematic changes to certain parts of the PDF, causing the PDF to render differently from the original. See also *PDF Transformation*.

6.9.1 Related Options

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

This flag starts page selection options, which are used to select pages from one or more input files to perform operations such as splitting, merging, and collating files.

Please see *Page Selection* for details about selecting pages.

See also `--split-pages`, `--collate`, *Page Ranges*.

`--collate[=n]`

This option causes **qpdf** to collate rather than concatenate pages specified with `--pages`. With a numeric parameter, collate in groups of *n*. The default is 1.

Please see *Page Selection* for additional details.

`--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.

Zero padding is added to all page numbers in file names so that all the numbers are the same length, which causes the output filenames to sort lexically in numerical order.

Page ranges are a single number in the case of single-page groups or two numbers separated by a dash otherwise.

Here are some examples. In these examples, `infile.pdf` has 12 pages.

- `qpdf --split-pages infile.pdf %d-out`: output files are `01-out` through `12-out` with no extension.
- `qpdf --split-pages=2 infile.pdf outfile.pdf`: output files are `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`. The extension `.else` is not treated in any special way regarding the placement of the number.

Note that outlines, threads, and other document-level 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 document-level data, you will have to run **qpdf** with the *--pages* option once for each page. Using *--split-pages* is much faster if you don't require the document-level data. A future version of qpdf may support preservation of some document-level information.

--overlay file [options] --

Overlay pages from another file on the output.

See *Overlay and Underlay* for details.

--underlay file [options] --

Underlay pages from another file on the output.

See *Overlay and Underlay* for 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. There is usually no reason to use this option unless you are working around a specific problem.

--flatten-annotations=parameter

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 value of *parameter* may be any of the following:

Table 4: Flatten Annotation Parameters

`all`	include all annotations that are not marked invisible or hidden
`print`	only include annotations that should appear when the page is printed
`screen`	omit annotations that should not appear on the screen

In a PDF file, interactive form fields have a value and, independently, a set of instructions, called an appearance, to render the filled-in field. If a form is filled in by a program that doesn't know how to update the appearances, they may become inconsistent with the fields' values. 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 for details.

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

Rotate the specified range of pages by the specified angle, which must be a multiple of 90 degrees.

The value of *angle* may be `0`, `90`, `180`, or `270`.

For a description of the syntax of *page-range*, see *Page Ranges*. If the page range is omitted, the rotation is applied to all pages.

If + is prepended to *angle*, the angle is added, so an angle of `+90` indicates a 90-degree clockwise rotation. If - is prepended, the angle is subtracted, so `-90` is a 90-degree counterclockwise rotation and is exactly the same as `+270`.

If neither + or - is prepended, the rotation angle is set exactly. You almost always want + or - since, without inspecting the actual PDF code, it is impossible to know whether a page that appears to be rotate is rotated "naturally" or has been rotated by specifying rotation. For example, if a page appears to contain a portrait-mode image rotated by 90 degrees so that the top of the image is on the right edge of the page, there is no way to tell by visual inspection whether the literal top of the image is the top of the page or whether the literal top of the image is the right edge and the page is already rotated in the PDF. Specifying a rotation angle of `-90` will produce an image that appears upright in either case. Use of absolute rotation angles should be reserved for cases in which you have specific knowledge about the way the PDF file is constructed.

Examples:

- `qpdf in.pdf out.pdf --rotate=+90:2,4,6 --rotate=+180:7-8`: rotate pages 2, 4, and 6 by 90 degrees clockwise from their original rotation
- `qpdf in.pdf out.pdf --rotate=+180`: rotate all pages by 180 degrees
- `qpdf in.pdf out.pdf --rotate=0`: force each page to be displayed in its natural orientation, which would undo the effect of any rotations previously applied in page metadata.

See also *--flatten-rotation*.

--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 usually no reason to do this, but it can be necessary before using the *--flatten-annotations* option. Here is a summary of the limitations.

- 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. **qpdf** does not know enough about fonts and encodings to correctly represent characters that fall outside of this range.
- For variable text fields where the default appearance stream specifies that the font should be auto-sized, a fixed font size is used rather than calculating the font size.
- 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.

Appearances generated by **qpdf** should be good enough for simple forms consisting of ASCII characters where the original file followed the PDF specification and provided template information for text field appearances. 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**. Most PDF viewers that support filling in of forms will generate appearance streams. Some of them will even do it for forms filled in with characters outside the original font's character range by embedding additional fonts as needed.

--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, 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.

--keep-inline-images

Prevent inline images from being included in image optimization done by *--optimize-images*.

--remove-page-labels

Exclude page labels (explicit page numbers) from the output file.

6.10 Encryption

This section describes the options used to create encrypted files. For other options related to encryption, see also *--decrypt* and *--copy-encryption*. For a more in-depth technical discussion of how PDF encryption works internally, see *PDF Encryption*.

To create an encrypted file, use

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

Either or both of *user-password* and *owner-password* may be empty strings. *key-length* may be `40`, `128`, or `256`. Encryption options are terminated by `--` by itself.

40-bit encryption is insecure, as is 128-bit encryption without AES. Use 256-bit encryption unless you have a specific reason to use an insecure format, such as testing or compatibility with very old viewers. You must use the *--allow-weak-crypto* flag to create encrypted files that use insecure cryptographic algorithms. The *--allow-weak-crypto* flag appears outside of `--encrypt ... --` (before `--encrypt` or after `--`).

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 one described in the PDF 2.0 specification. Using 128-bit encryption forces the PDF version to be at least 1.4, or if AES is used, 1.6. Using 40-bit encryption forces the PDF version to be at least 1.3.

When 256-bit encryption is used, PDF files with empty owner passwords are insecure. To create such files, you must specify the *--allow-insecure* option.

Available options vary by key length. Not all readers respect all restrictions. The default for each permission option is to be fully permissive. These restrictions may or may not be enforced by any particular reader. **qpdf** allows very granular setting of restrictions. Some readers may not recognize the combination of options you specify. If you specify certain combinations of restrictions and find a reader that doesn't seem to honor them as you expect, it is most likely

not a bug in **qpdf**. qpdf itself 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.

Here is a summary of encryption options. Details are provided in the help for each option.

Table 5: Options for 40-bit Encryption Only

`--annotate=[y|n]`	restrict comments, filling forms, and signing
`--extract=[y|n]`	restrict text/graphic extraction
`--modify=[y|n]`	restrict document modification
`--print=[y|n]`	restrict printing

Table 6: Options for 128-bit or 256-bit Encryption

`--accessibility=[y|n]`	restrict accessibility (usually ignored)
`--annotate=[y|n]`	restrict commenting/filling form fields
`--assemble=[y|n]`	restrict document assembly
`--extract=[y|n]`	restrict text/graphic extraction
`--form=[y|n]`	restrict filling form fields
`--modify-other=[y|n]`	restrict other modifications
`--modify=modify-opt`	control modify access by level
`--print=print-opt`	control printing access
`--cleartext-metadata`	prevent encryption of metadata

Table 7: Options for 128-bit Encryption Only

`--use-aes=[y|n]`	indicates whether to use AES encryption
`--force-V4`	forces use of V=4 encryption handler

Table 8: Options for 256-bit Encryption Only

`--force-R5`	forces use of deprecated `R=5` encryption algorithm
`--allow-insecure`	allow user password with empty owner password

Table 9: Values for *`print-opt`*

`none`	disallow printing
`low`	allow only low-resolution printing
`full`	allow full printing

Table 10: Values for *`modify-opt`*

`none`	allow no modifications
`assembly`	allow document assembly only
`form`	`assembly` permissions plus filling in form fields and signing
`annotate`	`form` permissions plus commenting and modifying forms
`all`	allow full document modification

6.10.1 Related Options

--accessibility=[y|n]

Enable/disable extraction of text for accessibility to visually impaired users. The default is to be fully permissive. The qpdf library disregards this field when AES is used with 128-bit encryption or when 256-bit encryption is used. You should never disable accessibility unless you are explicitly doing so for creating test files. The PDF spec says that conforming readers should disregard this permission and always allow accessibility.

This option is not available with 40-bit encryption.

--annotate=[y|n]

Enable/disable modifying annotations including making comments and filling in form fields. The default is to be fully permissive. For 128-bit and 256-bit encryption, this also enables editing, creating, and deleting form fields unless `--modify-other=n` or `--modify=none` is also specified.

--assemble=[y|n]

Enable/disable document assembly (rotation and reordering of pages). The default is to be fully permissive.

This option is not available with 40-bit encryption.

--extract=[y|n]

Enable/disable text/graphic extraction for purposes other than accessibility. The default is to be fully permissive.

--form=[y|n]

Enable/disable whether filling form fields is allowed even if modification of annotations is disabled. The default is to be fully permissive.

This option is not available with 40-bit encryption.

--modify-other=[y|n]

Enable/disable modifications not controlled by *--assemble*, *--annotate*, or *--form*. `--modify-other=n` is implied by any of the other *--modify* options except for `--modify=all`. The default is to be fully permissive.

This option is not available with 40-bit encryption.

--modify=modify-opt

For 40-bit files, *modify-opt* may only be `y` or `n` and controls all aspects of document modification. The default is to be fully permissive.

For 128-bit and 256-bit encryption, *modify-opt* values allow enabling and disabling levels of restriction in a manner similar to how some PDF creation tools do it:

Table 11: *modify-opt* for 128-bit and 256-bit Encryption

`none`	allow no modifications
`assembly`	allow document assembly only
`form`	`assembly` permissions plus filling in form fields and signing
`annotate`	`form` permissions plus commenting and modifying forms
`all`	allow full document modification (the default)

Modify options correspond to the more granular options as follows:

Table 12: Mapping *modify-opt* to Other Options

`none`	`--modify-other=n --annotate=n --form=n --assemble=n`
`assembly`	`--modify-other=n --annotate=n --form=n`
`form`	`--modify-other=n --annotate=n`
`annotate`	`--modify-other=n`
`all`	no other modify options (the default)

You can combine this option with the options listed above. If you do, later options override earlier options.

--print=print-opt

Control what kind of printing is allowed. The default is to be fully permissive. For 40-bit encryption, *print-opt* may only be `y` or `n` and enables or disables all printing. For 128-bit and 256-bit encryption, *print-opt* may have the following values:

Table 13: *print-opt* Values

`none`	disallow printing
`low`	allow low-resolution printing only
`full`	allow full printing (the default)

--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.

This option is not available with 40-bit encryption.

--use-aes=[y|n]

Enables/disables use of the more secure AES encryption with 128-bit encryption. Specifying `--use-aes=y` forces the PDF version to be at least 1.6. This option is only available with 128-bit encryption. The default is `n` for compatibility reasons. Use 256-bit encryption instead.

--allow-insecure

Allow creation of PDF files with 256-bit keys where the user password is non-empty and the owner password is empty. Files created in this way are insecure since they can be opened without a password, and restrictions will not be enforced. Users would ordinarily never want to create such files. If you are using qpdf to intentionally created strange files for testing (a valid use of qpdf!), this option allows you to create such insecure files. This option is only available with 256-bit encryption.

See *User and Owner Passwords* for a more technical discussion of this issue.

--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.

--force-R5

Use an undocumented, unsupported, deprecated encryption algorithm that existed only in Acrobat version IX. This option should not be used except for compatibility testing. If specified, qpdf sets the minimum version to 1.7 at extension level 3.

6.11 Page Selection

qpdf allows you to use the *--pages* option to split and merge PDF files by selecting pages from one or more input files.

Usage: `qpdf` *in.pdf* `--pages input-file [--password=`*password*`] [`*page-range*`] [...] --` *out.pdf*

Between `--pages` and the `--` that terminates pages option, repeat the following:

filename `[--password=`*password*`] [`*page-range*`]`

Notes:

- The password option is needed only for password-protected files. If you specify the same file more than once, you only need to supply the password the first time.
- The page range may be omitted. If omitted, all pages are included.
- Document-level information, such as outlines, tags, etc., is taken from the primary input file (in the above example, `in.pdf`) and is preserved in `out.pdf`. You can use `--empty` in place of an input file to start from an empty file and just copy pages equally from all files.
- You can use `.` as a shorthand for the primary input file, if not empty.

See *Page Ranges* for help on specifying a page range.

Use `--collate=n` to cause pages to be collated in groups of *n* pages (default 1) instead of concatenating the input. Note that the `--collate` appears outside of `--pages ... --` (before `--pages` or after `--`). Pages are pulled from each document in turn. When a document is out of pages, it is skipped. See examples below.

6.11.1 Examples

- Start with `in.pdf` and append all pages from `a.pdf` and the even pages from `b.pdf`, and write the output to `out.pdf`. Document-level information from `in.pdf` is retained. Note the use of `.` to refer to `in.pdf`.

```
qpdf in.pdf --pages . a.pdf b.pdf:even -- out.pdf
```

- Take all the pages from `a.pdf`, all the pages from `b.pdf` in reverse, and only pages 3 and 6 from `c.pdf` and write the result to `out.pdf`. Document-level metadata is discarded from all input files. The password `x` is used to open `b.pdf`.

```
qpdf --empty --pages a.pdf b.pdf --password=x z-1 c.pdf 3,6
```

- Scan a document with double-sided printing by scanning the fronts into `odd.pdf` and the backs into `even.pdf`. Collate the results into `all.pdf`. This takes the first page of `odd.pdf`, the first page of `even.pdf`, the second page of `odd.pdf`, the second page of `even.pdf`, etc.

```
qpdf --collate odd.pdf --pages . even.pdf -- all.pdf
  OR
qpdf --collate --empty --pages odd.pdf even.pdf -- all.pdf
```

- When collating, 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

- You can specify a numeric parameter to *--collate*. With `--collate=n`, pull groups of *n* pages from each file, as always, stopping when there are no more pages. For example, if you ran

```
qpdf --collate=2 --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
  - a.pdf page 2
  - b.pdf page 6
  - b.pdf page 5
  - c.pdf last page
  - a.pdf page 3
  - a.pdf page 4
  - b.pdf page 4
  - a.pdf page 5

- Take pages 1 through 5 from `file1.pdf` and pages 11 through 15 in reverse from `file2.pdf`, taking document-level metadata from `file2.pdf`.

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

- Here's a more contrived example. 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 without any shared data between the two copies of page 1, and if you wanted to drop document-level metadata but preserve encryption, you could run

```
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 separate files. (This is by design. See *Limitations* for a discussion.) These are all corner cases that most users should hopefully never have to be bothered with.

6.11.2 Limitations

With the exception of page labels (page numbers), **qpdf** doesn't yet have full support for handling document-level data as it relates to pages. Certain document-level features such as form fields, outlines (bookmarks), and article tags among others, are copied in their entirety from the primary input file. Starting with qpdf version 8.3, page labels are preserved from all files unless *--remove-page-labels* is specified.

It is expected that a future version of **qpdf** will have more complete and configurable behavior regarding document-level metadata. In the meantime, 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. If you don't want to preserve the primary file's metadata, use *--empty* as the primary input file.

Visit qpdf issues labeled with "pages" or look at the TODO file in the qpdf source distribution for some of the ideas.

Prior to **qpdf** version 8.4, it was not possible to specify the same page from the same file directly more than once, and a workaround of specifying the same file in more than one way was required. Version 8.4 removes this limitation, but when the same page is copied more than once, all its data is shared between the pages. Sometimes this is fine, but sometimes it may not work correctly, particularly if there are form fields or you intend to perform other modifications on one of the pages. A future version of qpdf should address this more completely. You can work around this by specifying the same 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 not share any objects in common. This includes fonts, images, and anything else the page references.

6.12 Overlay and Underlay

You can use **qpdf** 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 options 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, but you can only specify each option at most one time.

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:

--to=page-range

Specify a page range (see *Page Ranges*) that 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]

Specify a page range that indicates which pages in the overlay/underlay file will be used for overlay or underlay. If not specified, all pages will be used. The "from" pages are used until they are exhausted, after which any pages specified with *--repeat* are used. If you are using the *--repeat* option, you can use `--from=` to provide an empty set of "from" pages.

This Can be left empty by omitting *page-range*

--repeat=page-range

Specify an optional page range that indicates which pages in the overlay/underlay file will be repeated after the "from" pages are used up. If you want to apply a repeat a range of pages starting with the first page of output, you can explicitly use `--from=`.

6.12.1 Examples

- 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.

```
qpdf in.pdf --overlay o.pdf --to=1-5 --from=1-3 --repeat=4 -- out.pdf
```

- Underlay page 1 of `footer.pdf` on all odd output pages, and underlay page 2 of `footer.pdf` on all even output pages.

```
qpdf in.pdf --underlay footer.pdf --from= --repeat=1,2 -- out.pdf
```

- Combine two files and overlay the single page from watermark.pdf on the result.

```
qpdf --empty --pages a.pdf b.pdf -- \
     --overlay watermark.pdf --from= --repeat=1 -- out.pdf
```

6.13 Embedded Files/Attachments

It is possible to list, add, or delete embedded files (also known as attachments) and to copy attachments from other files. See also *--list-attachments* and *--show-attachment*.

6.13.1 Related Options

--add-attachment file [options] --

This flag starts add attachment options, which are used to add attachments to a file.

The `--add-attachment` flag and its options may be repeated to add multiple attachments. Please see *Options for Adding Attachments* for additional details.

--copy-attachments-from file [options] --

This flag starts copy attachment options, which are used to copy attachments from other files.

The `--copy-attachments-from` flag and its options may be repeated to copy attachments from multiple files. Please see *Options for Copying Attachments* for additional details.

--remove-attachment=key

Remove the specified attachment. This doesn't only remove the attachment from the embedded files table but also clears out the file specification to ensure that the attachment is actually not present in the output file. That means that any potential internal links to the attachment will be broken. Run with *--verbose* to see status of the removal. Use *--list-attachments* to find the attachment key. This option may be repeated to remove multiple attachments.

6.13.2 PDF Date Format

When a date is required, the date should conform to the PDF date format specification, which is `D:yyyymmddhhmmssz` where *z* is either literally upper case `Z` for UTC or a timezone offset in the form *-hh'mm'* or *+hh'mm'*. Negative timezone offsets indicate time before UTC. Positive offsets indicate how far after. For example, US Eastern Standard Time (America/New_York) is `-05'00'`, and Indian Standard Time (Asia/Calcutta) is `+05'30'`.

Table 14: PDF Date Examples

`D:20210207161528-05'00'`	February 7, 2021 at 4:15:28 p.m.
`D:20210207211528Z`	February 7, 2021 at 21:15:28 UTC

6.13.3 Options for Adding Attachments

These options are valid between `--add-attachment` and `--`.

`--key=key`

Specify the key to use for the attachment in the embedded files table. It defaults to the last element of the attached file's filename. For example, if you say `--add-attachment /home/user/image.png`, the default key will be just `image.png`.

`--filename=name`

Specify the filename to be used for the attachment. This is what is usually displayed to the user and is the name most graphical PDF viewers will use when saving a file. It defaults to the last element of the attached file's filename. For example, if you say `--add-attachment /home/user/image.png`, the default key will be just `image.png`.

`--creationdate=date`

Specify the attachment's creation date in PDF format; defaults to the current time. See *PDF Date Format* for information about the date format.

`--moddate=date`

Specify the attachment's modification date in PDF format; defaults to the current time. See *PDF Date Format* for information about the date format.

`--mimetype=type/subtype`

Specify the mime type for the attachment, such as `text/plain`, `application/pdf`, `image/png`, etc. The qpdf library does not automatically determine the mime type. In a UNIX-like environment, the **`file`** command can often provide this information. In MacOS, you can use `file -I filename`. In Linux, it's `file -i filename`.

Implementation note: the mime type appears in a field called `/Subtype` in the PDF file, but that field actually includes the full type and subtype of the mime type. This is because `/Type` already means something else in PDF.

`--description="text"`

Supply descriptive text for the attachment, displayed by some PDF viewers.

`--replace`

Indicate that any existing attachment with the same key should be replaced by the new attachment. Otherwise, **`qpdf`** gives an error if an attachment with that key is already present.

6.13.4 Options for Copying Attachments

Options in this section are valid between *--copy-attachments-from* and --.

--prefix=prefix

Only required if the file from which attachments are being copied has attachments with keys that conflict with attachments already in the file. In this case, the specified prefix will be prepended to each key. This affects only the key in the embedded files table, not the file name. The PDF specification doesn't preclude multiple attachments having the same file name.

6.14 PDF Inspection

These options provide tools for inspecting PDF files. When any of the options in this section are specified, no output file may be given.

6.14.1 Related Options

--is-encrypted

Silently exit with a code indicating the file's encryption status:

Table 15: Exit Codes for --is-encrypted

0	the file is encrypted
1	not used
2	the file is not encrypted

This option can be used for password-protected files even if you don't know the password.

This option 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 a code indicating the file's password status:

Table 16: Exit Codes for --requires-password

0	a password, other than as supplied, is required
1	not used
2	the file is not encrypted
3	the file is encrypted, and correct password (if any) has been supplied

Use with the *--password* option to specify the password to test.

The choice of exit status 0 to mean that a password is required is to enable code like

```
if [ qpdf --requires-password file.pdf ]; then
    # prompt for password
fi
```

If a password is supplied with *--password*, that password is used to open the file just as with any normal invocation of **qpdf**. That means that using this option with *--password* 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.

--check

Check the file's structure as well as encryption, linearization, and encoding of stream data, and write information about the file to standard output. An exit status of 0 indicates syntactic correctness of the PDF file. Note that --check writes nothing to standard error when everything is valid, so if you are using this to programmatically validate files in bulk, it is safe to run without output redirected to /dev/null and just check for a 0 exit code.

A file for which --check reports no errors may still have errors in stream data content or may contain constructs that don't conform to the PDF specification, but it should be syntactically valid. If --check reports 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. 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.

See also *Exit Status*.

--show-encryption

This option shows document encryption parameters. It also shows the document's user password if the owner password is given and the file was encrypted using older encryption formats that allow user password recovery. (See *PDF Encryption* for a technical discussion of this feature.) The output of --show-encryption is included in the output of *--check*.

--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 parameter 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. See *PDF Encryption* for a technical discussion.

--check-linearization

Check to see whether a file is linearized and, if so, whether the linearization hint tables are correct. qpdf does not check all aspects of linearization. A linearized PDF file with linearization errors that is otherwise correct is almost always readable by a PDF viewer. As such, "errors" in PDF linearization are treated by **qpdf** as warnings.

--show-linearization

Check and display all data in the linearization hint tables.

--show-xref

Show the contents of the cross-reference table or stream in a human-readable form. The cross-reference data gives the offset of regular objects and the object stream ID and 0-based index within the object stream for compressed objects. This is especially useful for files with cross-reference streams, which are stored in a binary format. If the file is invalid and cross reference table reconstruction is performed, this option will show the information in the reconstructed table.

--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 with *--show-object*, if the object is a stream, write the raw (compressed) binary stream data to standard output instead of the object's contents. Avoid combining this with other inspection options to avoid commingling the stream data with other output. See also *--filtered-stream-data*.

--filtered-stream-data

When used with *--show-object*, if the object is a stream, write the filtered (uncompressed, potentially binary) stream data to standard output instead of the object's contents. If the stream is filtered using filters that qpdf does not support, an error will be issued. This option acts as if `--decode-level=all` was specified (see *--decode-level*), so it will uncompress images compressed with supported lossy compression schemes. Avoid combining this with other inspection options to avoid commingling the stream data with other output.

This option may be combined with *--normalize-content*. If you do this, qpdf will attempt to run content normalization even if the stream is not a content stream, which will probably produce unusable results.

See also *--raw-stream-data*.

--show-npages

Print 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

Show 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. See also *--with-images*.

--with-images

When used with *--show-pages*, also shows the object and generation numbers for the image objects on each page.

--list-attachments

Show the *key* and stream number for each embedded file. With *--verbose*, additional information, including preferred file name, description, dates, and more are also displayed. The key is usually but not always equal to the file name and is needed by some of the other options. See also *Embedded Files/Attachments*. Note that this option displays dates in PDF timestamp syntax. When attachment information is included in json output in the `"attachments"` key (see *--json*), dates are shown (just within that object) in ISO-8601 format.

--show-attachment=key

Write the contents of the specified attachment to standard output as binary data. The key should match one of the keys shown by *--list-attachments*. If this option is given more than once, only the last attachment will be shown. See also *Embedded Files/Attachments*.

6.15 JSON Options

It is possible to view information about PDF files in a JSON format. See *qpdf JSON* for details about the qpdf JSON format.

6.15.1 Related Options

--json[=version]

Generate a JSON representation of the file. This is described in depth in *qpdf JSON*. The version parameter can be used to specify which version of the qpdf JSON format should be output. The version number be a number or `latest`. The default is `latest`. As of qpdf 11, the latest version is 2. If you have code that reads qpdf JSON output, you can tell what version of the JSON output you have from the `"version"` key in the output. Use the *--json-help* option to get a description of the JSON object.

Starting with qpdf 11, when this option is specified, an output file is optional (for backward compatibility) and defaults to standard output. You may specify an output file to write the JSON to a file rather than standard output. (Example: `qpdf --json in.pdf out.json`)

Stream data is only included if *--json-output* is specified or if a value other than none is passed to *--json-stream-data*.

--json-help[=version]

Describe the format of the corresponding version of JSON output by writing to standard output a JSON object with the same structure as the JSON generated by qpdf. In the output written by --json-help, each key's value is a description of the key. The specific contract guaranteed by qpdf in its JSON representation is explained in more detail in the *qpdf JSON*. The default version of help is version 2, as with the *--json* flag.

--json-key=key

This option is repeatable. If given, only the specified top-level keys will be included in the JSON output. Otherwise, all keys will be included. If not given, all keys will be included, unless *--json-output* was specified, in which case, only the "qpdf" key will be included by default. If *--json-output* was not given, the version and parameters keys will always appear in the output.

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

This option is repeatable. If given, only specified objects will be shown in the objects dictionary in the JSON output. Otherwise, all objects will be shown. See *qpdf JSON* for details about the qpdf JSON format.

--json-stream-data={none|inline|file}

When used with *--json*, this option controls whether streams in JSON output should be omitted, written inline (base64-encoded) or written to a file. If file is chosen, the file will be the name of the output file appended with *-nnn* where *nnn* is the object number. The stream data file prefix can be overridden with *--json-stream-prefix*. The default value is none, except when *--json-output* is specified, in which case the default is inline.

--json-stream-prefix=file-prefix

When used with --json-stream-data=file, --json-stream-data=file-prefix sets the prefix for stream data files, overriding the default, which is to use the output file name. Whatever is given here will be appended with *-nnn* to create the name of the file that will contain the data for the stream stream in object *nnn*.

--json-output[=version]

Implies *--json* at the specified version. This option changes several default values, all of which can be overridden by specifying the stated option:

- The default value for *--json-stream-data* changes from none to inline.
- The default value for *--decode-level* changes from generalized to none.
- By default, only the "qpdf" key is included in the JSON output, but you can add additional keys with *--json-key*.
- The "version" and "parameters" keys will be excluded from the JSON output.

If you want to look at the contents of streams easily as you would in QDF mode (see *QDF Mode*), you can use --decode-level=generalized and --json-stream-data=file for a convenient way to do that.

--json-input

Treat the input file as a JSON file in qpdf JSON format. The input file must be complete and include all stream data. The JSON version must be at least 2. All top-level keys are ignored except for "qpdf". For information about converting between PDF and JSON, please see *qpdf JSON*.

--update-from-json=qpdf-json-file

This option updates a PDF file from the specified qpdf JSON file. For a information about how to use this option, please see *qpdf JSON*.

6.16 Options for Testing or Debugging

The options below are useful when writing automated test code that includes files created by qpdf or when testing qpdf itself. When changes are made to qpdf, care is taken to avoid gratuitously changing the output of PDF files. This is to make it easier to do direct comparisons in test suites with files created by qpdf. However, there are no guarantees that the PDF output won't change such as in the event of a bug fix or feature enhancement to some aspect of the output that qpdf creates.

6.16.1 Idempotency

Note about idempotency of byte-for-byte content: there is no expectation that qpdf is idempotent in the general case. In other words, there is no expectation that, when qpdf is run on its own output, it will create *byte-for-byte* identical output, even though it will create semantically identical files. There are a variety of reasons for this including document ID generation, which includes a random element, as well as the interaction of stream length encoding with dictionary key sorting.

It is possible to get idempotent behavior by using the *--static-id* or *--deterministic-id* option with qpdf and running it *three* times so that you are processing the output of qpdf on its own previous output. For example, in this sequence of commands:

```
qpdf any-file.pdf 1.pdf
qpdf --static-id 1.pdf 2.pdf
qpdf --static-id 2.pdf 3.pdf
```

the files `2.pdf` and `3.pdf` should be *byte-for-byte* identical. The qpdf test suite relies on this behavior. See also *--static-aes-iv*.

6.16.2 Related Options

--static-id

Use a fixed value for the document ID (`/ID` in the trailer). **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

Use 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.** When combined with *--static-id* and using the three-step process described in *Idempotency*, it is possible to create byte-for-byte idempotent output with PDF files that use 256-bit encryption to assist with creating reproducible test suites.

--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, allowing inspection of the file before values calculated in pass 1 are inserted into the file for pass 2.

--test-json-schema

This is used by qpdf's test suite to check consistency between the output of `qpdf --json` and the output of `qpdf --json-help`. This option causes an extra copy of the generated JSON to appear in memory and is therefore unsuitable for use with large files. This is why it's also not on by default.

`--report-memory-usage`

This is used by qpdf's performance test suite to report the maximum amount of memory used in supported environments.

6.17 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 interacting 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 no 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 passwords 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. That flag bypasses the normal process of going from password to encryption key entirely, allowing the raw encryption key to be specified directly. That behavior is useful for forensic purposes or for brute-force recovery of files with unknown passwords and has nothing to do with the document's actual passwords.

CHAPTER

SEVEN

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.

For another way to work with PDF files in an editor, see *qpdf JSON*. Using qpdf JSON format allows you to edit the PDF file semantically without having to be concerned about PDF syntax. However, QDF files are actually valid PDF files, so the feedback cycle may be faster if previewing with a PDF reader. Also, since QDF files are valid PDF, you can experiment with all aspects of the PDF file, including syntax.

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

EIGHT

USING THE QPDF LIBRARY

8.1 Using QPDF from C++

The source tree for the qpdf package has an `examples` directory that contains a few example programs. The `libqpdf/QPDFJob.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.

qpdf installs a `pkg-config` configuration with package name `libqpdf` and a `cmake` configuration with package name `qpdf`. The `libqpdf` target is exported in the `qpdf::` namespace. The following is an example of a `CMakeLists.txt` file for a single-file executable that links with qpdf:

```
cmake_minimum_required(VERSION 3.16)
project(some-application LANGUAGES CXX)
find_package(qpdf)
add_executable(some-application some-application.cc)
target_link_libraries(some-application qpdf::libqpdf)
```

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.

8.2 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 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 11.0.0, the `qpdf`
command-line tool can produce an unambiguous JSON representation of a PDF file and can also create or update PDF files using this JSON representation. qpdf versions from 8.3.0 through 10.6.3 had a more limited JSON output format. The qpdf JSON format makes it possible to inspect and modify the structure of a PDF file down to the object level from the command-line or from any language that can handle JSON data. Please see *qpdf JSON* for details.

Wrappers
The qpdf Wiki contains a list of Wrappers around qpdf. These may have varying degrees of functionality or completeness. If you know of (or have written) a wrapper that you'd like include, open an issue at https://github.com/qpdf/qpdf/issues/new and ask for it to be added to the list.

8.3 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 or CLI output messages. Patches or bug reports are welcome for any continuing issues with Unicode file names in Windows.

CHAPTER

NINE

WEAK CRYPTOGRAPHY

For help with compiler errors in qpdf 11.0 or newer, see *API-Breaking Changes in qpdf 11.0*.

Since 2006, the PDF specification has offered ways to create encrypted PDF files without using weak cryptography, though it took a few years for many PDF readers and writers to catch up. It is still necessary to support weak encryption algorithms to read encrypted PDF files that were created using weak encryption algorithms, including all PDF files created before the modern formats were introduced or widely supported.

Starting with version 10.4, qpdf began taking steps to reduce the likelihood of a user *accidentally* creating PDF files with insecure cryptography but will continue to allow creation of such files indefinitely with explicit acknowledgment. The restrictions on use of weak cryptography were made stricter with qpdf 11.

9.1 Definition of Weak Cryptographic Algorithm

We divide weak cryptographic algorithms into two categories: weak encryption and weak hashing. Encryption is encoding data such that a key of some sort is required to decode it. Hashing is creating a short value from data in such a way that it is extremely improbable to find two documents with the same hash (known has a hash collision) and extremely difficult to intentionally create a document with a specific hash or two documents with the same hash.

When we say that an encryption algorithm is weak, we either mean that a mathematical flaw has been discovered that makes it inherently insecure or that it is sufficiently simple that modern computer technology makes it possible to use "brute force" to crack. For example, when 40-bit keys were originally introduced, it wasn't practical to consider trying all possible keys, but today such a thing is possible.

When we say that a hashing algorithm is weak, we mean that, either because of mathematical flaw or insufficient complexity, it is computationally feasible to intentionally construct a hash collision.

While weak encryption should always be avoided, there are cases in which it is safe to use a weak hashing algorithm when security is not a factor. For example, a weak hashing algorithm should not be used as the only mechanism to test whether a file has been tampered with. In other words, you can't use a weak hash as a digital signature. There is no harm, however, in using a weak hash as a way to sort or index documents as long as hash collisions are tolerated. It is also common to use weak hashes as checksums, which are often used a check that a file wasn't damaged in transit or storage, though for true integrity, a strong hash would be better.

Note that qpdf must always retain support for weak cryptographic algorithms since this is required for reading older PDF files that use it. Additionally, qpdf will always retain the ability to create files using weak cryptographic algorithms since, as a development tool, qpdf explicitly supports creating older or deprecated types of PDF files since these are sometimes needed to test or work with older versions of software. Even if other cryptography libraries drop support for RC4 or MD5, qpdf can always fall back to its internal implementations of those algorithms, so they are not going to disappear from qpdf.

9.2 Uses of Weak Encryption in qpdf

When PDF files are encrypted using 40-bit encryption or 128-bit encryption without AES, then the weak *RC4* algorithm is used. You can avoid using weak encryption in qpdf by always using 256-bit encryption. Unless you are trying to create files that need to be opened with PDF readers from before about 2010 (by which time most readers had added support for the stronger encryption algorithms) or are creating insecure files explicitly for testing or some similar purpose, there is no reason to use anything other than 256-bit encryption.

By default, qpdf refuses to write a file that uses weak encryption. You can explicitly allow this by specifying the `--allow-weak-crypto` option.

In qpdf 11, all library methods that could potentially cause files to be written with weak encryption were deprecated, and methods to enable weak encryption were either given explicit names indicating this or take required arguments to enable the insecure behavior.

There is one exception: when encryption parameters are copied from the input file or another file to the output file, there is no prohibition or even warning against using insecure encryption. The reason is that many qpdf operations simply preserve whatever encryption is there, and requiring confirmation to *preserve* insecure encryption would cause qpdf to break when non-encryption-related operations were performed on files that happened to be encrypted. Failing or generating warnings in this case would likely have the effect of making people use the `--allow-weak-crypto` option blindly, which would be worse than just letting those files go so that explicit, conscious selection of weak crypto would be more likely to be noticed. Why, you might ask, does this apply to `--copy-encryption` as well as to the default behavior preserving encryption? The answer is that `--copy-encryption` works with an unencrypted file as input, which enables workflows where one may start with a file, decrypt it *just in case*, perform a series of operations, and then reapply the original encryption, *if any*. Also, one may have a template used for encryption that one may apply to a variety of output files, and it would be annoying to be warned about it for every output file.

9.3 Uses of Weak Hashing In QPDF

The PDF specification makes use the weak *MD5* hashing algorithm in several places. While it is used in the encryption algorithms, breaking MD5 would not be adequate to crack an encrypted file when 256-bit encryption is in use, so using 256-bit encryption is adequate for avoiding the use of MD5 for anything security-sensitive.

MD5 is used in the following non-security-sensitive ways:

- Generation of the document ID. The document ID is an input parameter to the document encryption but is not itself considered to be secure. They are supposed to be unique, but they are not tamper-resistent in non-encrypted PDF files, and hash collisions must be tolerated.

  The PDF specification recommends but does not require the use of MD5 in generation of document IDs. Usually there is also a random component to document ID generation. There is a qpdf-specific feature of generating a *deterministic ID* (see `--deterministic-id`) which also uses MD5. While it would certainly be possible to change the deterministic ID algorithm to not use MD5, doing so would break all previous deterministic IDs (which would render the feature useless for many cases) and would offer very little benefit since even a securely generated document ID is not itself a security-sensitive value.

- Checksums in embedded file streams – the PDF specification specifies the use of MD5.

It is therefore not possible completely avoid the use of MD5 with qpdf, but as long as you are using 256-bit encryption, it is not used in a security-sensitive fashion.

9.4 API-Breaking Changes in qpdf 11.0

In qpdf 11, several deprecated functions and methods were removed. These methods provided an incomplete API. Alternatives were added in qpdf 8.4.0. The removed functions are

- C API: `qpdf_set_r3_encryption_parameters`, `qpdf_set_r4_encryption_parameters`, `qpdf_set_r5_encryption_parameters`, `qpdf_set_r6_encryption_parameters`
- `QPDFWriter`: overloaded versions of these methods with fewer arguments: `setR3EncryptionParameters`, `setR4EncryptionParameters`, `setR5EncryptionParameters`, and `setR6EncryptionParameters`

Additionally, remaining functions/methods had their names changed to signal that they are insecure and to force developers to make a decision. If you intentionally want to continue to use insecure cryptographic algorithms and create insecure files, you can change your code just add `_insecure` or `Insecure` to the end of the function as needed. (Note the disappearance of `2` in some of the C functions as well.) Better, you should migrate your code to use more secure encryption as documented in `QPDFWriter.hh`. Use the `R6` methods (or their corresponding C functions) to create files with 256-bit encryption.

Table 1: Renamed Functions

Old Name	New Name
qpdf_set_r2_encryption_parameters	qpdf_set_r2_encryption_parameters_insecure
qpdf_set_r3_encryption_parameters2	qpdf_set_r3_encryption_parameters_insecure
qpdf_set_r4_encryption_parameters2	qpdf_set_r2_encryption_parameters_insecure
QPDFWriter::setR2EncryptionParameters	QPDFWriter::setR2EncryptionParametersInsecure
QPDFWriter::setR3EncryptionParameters	QPDFWriter::setR3EncryptionParametersInsecure
QPDFWriter::setR4EncryptionParameters	QPDFWriter::setR4EncryptionParametersInsecure

CHAPTER

TEN

QPDF JSON

10.1 Overview

Beginning with qpdf version 11.0.0, the qpdf library and command-line program can produce a JSON representation of a PDF file. qpdf version 11 introduces JSON format version 2. Prior to qpdf 11, versions 8.3.0 onward had a more limited JSON representation accessible only from the command-line. For details on what changed, see *Changes from JSON v1 to v2*. The rest of this chapter documents qpdf JSON version 2.

Please note: this chapter discusses *qpdf JSON format*, which represents the contents of a PDF file. This is distinct from the *QPDFJob JSON format* which provides a higher-level interface interacting with qpdf the way the command-line tool does. For information about that, see *QPDFJob: a Job-Based Interface*.

The qpdf JSON format is specific to qpdf. The `--json` command-line flag causes creation of a JSON representation the objects in a PDF file along with JSON-formatted summaries of other information about the file. This functionality is built into `QPDFJob` and can be accessed from the `qpdf` command-line tool or from the `QPDFJob` C or C++ API.

Starting with qpdf JSON version 2, from qpdf 11.0.0, the JSON output includes an unambiguous and complete representation of the PDF objects and header. The information without the JSON-formatted summaries of other information is also available using the `QPDF::writeJSON` method.

By default, stream data is omitted from the JSON data, but it can be included by specifying the `--json-stream-data` option. With stream data included, the generated JSON file completely represents a PDF file. You can think of this as using JSON as an *alternative syntax* for representing a PDF file. Using qpdf JSON, it is possible to convert a PDF file to JSON, manipulate the structure or contents of the objects at a low level, and convert the results back to a PDF file. This functionality can be accessed from the command-line with the `--json-input`, and `--update-from-json` flags, or from the API using the `QPDF::createFromJSON`, and `QPDF::updateFromJSON` methods. The `--json-output` flag changes a handful of defaults so that the resulting JSON is as close as possible to the original input and is ready for being converted back to PDF.

The qpdf JSON data includes unreferenced objects. This may be addressed in a future version of qpdf. For now, that means that certain objects that are not useful in the JSON representation are included. This includes linearization and encryption dictionaries, linearization hint streams, object streams, and the cross-reference (xref) stream associated with the trailer dictionary where applicable. For the best experience with qpdf JSON, you can run the file through qpdf first to remove encryption, linearization, and object streams. For example:

```
qpdf --decrypt --object-streams=disable in.pdf out.pdf
qpdf --json-output out.pdf out.json
```

10.2 JSON Terminology

Notes about terminology:

- In JavaScript and JSON, that thing that has keys and values is typically called an *object*.
- In PDF, that thing that has keys and values is typically called a *dictionary*. An *object* is a PDF object such as integer, real, boolean, null, string, array, dictionary, or stream.
- Some languages that use JSON call an *object* a *dictionary*, a *map*, or a *hash*.
- Sometimes, it's called on *object* if it has fixed keys and a *dictionary* if it has variable keys.

This manual is not entirely consistent about its use of *dictionary* vs. *object* because sometimes one term or another is clearer in context. Just be aware of the ambiguity when reading the manual. We frequently use the term *dictionary* to refer to a JSON object because of the consistency with PDF terminology, particular when referring to a dictionary that contains information PDF objects.

10.3 What qpdf JSON is not

Please note that qpdf JSON offers a convenient syntax for manipulating PDF files at a low level using JSON syntax. JSON syntax is much easier to work with than native PDF syntax, and there are good JSON libraries in virtually every commonly used programming language. Working with PDF objects in JSON removes the need to worry about stream lengths, cross reference tables, and PDF-specific representations of Unicode or binary strings that appear outside of content streams. It does not eliminate the need to understand the semantic structure of PDF files. Working with qpdf JSON still requires familiarity with the PDF specification.

In particular, qpdf JSON *does not* provide any of the following capabilities:

- Text extraction. While you could use qpdf JSON syntax to navigate to a page's content streams and font structures, text within pages is still encoded using PDF syntax within content streams, and there is no assistance for text extraction.
- Reflowing text, document structure. qpdf JSON does not add any new information or insight into the content of PDF files. If you have a PDF file that lacks any structural information, qpdf JSON won't help you solve any of those problems.

This is what we mean when we say that JSON provides a *alternative syntax* for working with PDF data. Semantically, it is identical to native PDF.

10.4 qpdf JSON Format

This section describes how qpdf represents PDF objects in JSON format. It also describes how to work with qpdf JSON to create or modify PDF files.

10.4.1 qpdf JSON Object Representation

This section describes the representation of PDF objects in qpdf JSON version 2. An example appears in *qpdf JSON Example*.

PDF objects are represented within the "qpdf" entry of a qpdf JSON file. The "qpdf" entry is a two-element array. The first element is a dictionary containing header-like information about the file such as the PDF version. The second element is a dictionary containing all the objects in the PDF file. We refer to this as the *objects dictionary*.

The first element contains the following keys:

- "jsonversion" – a number indicating the JSON version used for writing. This will always be 2.
- "pdfversion" – a string containing PDF version as indicated in the PDF header (e.g. "1.7", "2.0")
- pushedinheritedpageresources – a boolean indicating whether the library pushed inherited resources down to the page level. Certain library calls cause this to happen, and qpdf needs to know when reading a JSON file back in whether it should do this as it may cause certain objects to be renumbered. This field is ignored when *--update-from-json* was not given.
- calledgetallpages – a boolean indicating whether getAllPages was called prior to writing the JSON output. This method causes page tree repair to occur, which may renumber some objects (in very rare cases of corrupted page trees), so qpdf needs to know this information when reading a JSON file back in. This field is ignored when *--update-from-json* was not given.
- "maxobjectid" – a number indicating the object ID of the highest numbered object in the file. This is provided to make it easier for software that wants to add new objects to the file as you can safely start with one above that number when creating new objects. Note that the value of "maxobjectid" may be higher than the actual maximum object that appears in the input PDF since it takes into consideration any dangling indirect object references from the original file. This prevents you from unwittingly creating an object that doesn't exist but that is referenced, which may have unintended side effects. (The PDF specification explicitly allows dangling references and says to treat them as nulls. This can happen if objects are removed from a PDF file.)

The second element is the objects dictionary. Each key in the objects dictionary is either "trailer" or a string of the form "obj:*O G* R" where *O* and *G* are the object and generation numbers and R is the literal string R. This is the PDF syntax for the indirect object reference prepended by obj:. The value, representing the object itself, is a JSON object whose structure is described below.

Top-level Stream Objects

Stream objects are represented as a JSON object with the single key "stream". The stream object has a key called "dict" whose value is the stream dictionary as an object value (described below) with the "/Length" key omitted. Other keys are determined by the value for json stream data (*--json-stream-data*, or a parameter of type qpdf_json_stream_data_e) as follows:

- none: stream data is not represented; no other keys are present specified.
- inline: the stream data appears as a base64-encoded string as the value of the "data" key
- file: the stream data is written to a file, and the path to the file is stored in the "datafile" key. A relative path is interpreted as relative to the current directory when qpdf is invoked.

Keys other than "dict", "data", and "datafile" are ignored. This is primarily for future compatibility in case a newer version of qpdf includes additional information.

As with the native PDF representation, the stream data must be consistent with whatever filters and decode parameters are specified in the stream dictionary.

Top-level Non-stream Objects

Non-stream objects are represented as a dictionary with the single key "value". Other keys are ignored for future compatibility. The value's structure is described in "Object Values" below.

Note: in files that use object streams, the trailer "dictionary" is actually a stream, but in the JSON representation, the value of the "trailer" key is always written as a dictionary (with a "value" key like other non-stream objects). There will also be a a stream object whose key is the object ID of the cross-reference stream, even though this stream will generally be unreferenced. This makes it possible to assume "trailer" points to a dictionary without having to consider whether the file uses object streams or not. It is also consistent with how QPDF::getTrailer behaves in the C++ API.

Object Values

Within "value" or "stream"."dict", PDF objects are represented as follows:

- Objects of type Boolean or null are represented as JSON objects of the same type.
- Objects that are numeric are represented as numeric in the JSON without regard to precision. Internally, qpdf stores numeric values as strings, so qpdf will preserve arbitrary precision numerical values when reading and writing JSON. It is likely that other JSON readers and writers will have implementation-dependent ways of handling numerical values that are out of range.
- Name objects are represented as JSON strings that start with / and are followed by the PDF name in canonical form with all PDF syntax resolved. For example, the name whose canonical form (per the PDF specification) is text/plain would be represented in JSON as "/text/plain" and in PDF as "/text#2fplain".
- Indirect object references are represented as JSON strings that look like a PDF indirect object reference and have the form "*O G* R" where *O* and *G* are the object and generation numbers and R is the literal string R. For example, "3 0 R" would represent a reference to the object with object ID 3 and generation 0.
- PDF strings are represented as JSON strings in one of two ways:
  - "u:utf8-encoded-string": this format is used when the PDF string can be unambiguously represented as a Unicode string and contains no unprintable characters. This is the case whether the input string is encoded as UTF-16, UTF-8 (as allowed by PDF 2.0), or PDF doc encoding. Strings are only represented this way if they can be encoded without loss of information.
  - "b:hex-string": this format is used to represent any binary string value that can't be represented as a Unicode string. hex-string must have an even number of characters that range from a through f, A through F, or 0 through 9.

  qpdf writes empty strings as "u:", but both "b:" and "u:" are valid representations of the empty string.

  There is full support for UTF-16 surrogate pairs. Binary strings encoded with "b:..." are the internal PDF representations. As such, the following are equivalent:
  - "u:\ud83e\udd54" – representation of U+1F954 as a surrogate pair in JSON syntax
  - "b:FEFFD83EDD54" – representation of U+1F954 as the bytes of a UTF-16 string in PDF syntax with the leading FEFF indicating UTF-16
  - "b:efbbbff09fa594" – representation of U+1F954 as the bytes of a UTF-8 string in PDF syntax (as allowed by PDF 2.0) with the leading EF, BB, BF sequence (which is just UTF-8 encoding of FEFF).
  - A JSON string whose contents are u: followed by the UTF-8 representation of U+1F954. This is the potato emoji. Unfortunately, I am not able to render it in the PDF version of this manual.
- PDF arrays are represented as JSON arrays of objects as described above
- PDF dictionaries are represented as JSON objects whose keys are the string representations of names and whose values are representations of PDF objects.

Note that writing JSON output is done by QPDF, not QPDFWriter. As such, none of the things QPDFWriter does apply. This includes recompression of streams, renumbering of objects, removal of unreferenced objects, encryption, decryption, linearization, QDF mode, etc. See *Writing PDF Files* for a more in-depth discussion. This has a few noteworthy implications:

- Decryption is handled transparently by qpdf. As there are no QPDF APIs, even internal to the library, that allow retrieval of encrypted data in its raw, encrypted form, qpdf JSON always includes decrypted data. It is possible that a future version of qpdf may allow access to raw, encrypted string and stream data.
- Objects that are related to a PDF file's structure, rather than its content, are included in the JSON output, even though they are not particularly useful. In a future version of qpdf, this may be fixed, and the *--preserve-unreferenced* flag may be able to be used to get the existing behavior. For now, to avoid this, run the file through `qpdf --decrypt --object-streams=disable in.pdf out.pdf` to generate a new PDF file that contains no unreferenced or structural objects.
  - Linearized PDF files include a linearization dictionary which is not referenced from any other object and which references the linearization hint stream by offset. The JSON from a linearized PDF file contains both of these objects, even though they are not useful in the JSON. Offset information is not represented in the JSON, so there's no way to find the linearization hint stream from the JSON. If a new PDF is created from JSON that was written, the objects will be read back in but will just be unreferenced objects that will be ignored by `QPDFWriter` when the file is rewritten.
  - The JSON from a file with object streams will include the original object stream and will also include all the objects in the stream as top-level objects.
  - In files with object streams, the trailer "dictionary" is a stream. In qpdf JSON files, the `"trailer"` key will contain a dictionary with all the keys in it relating to the stream, and the stream will also appear as an unreferenced object.
  - Encrypted files are decrypted, but the encryption dictionary still appears in the JSON output.

10.4.2 qpdf JSON Example

The JSON below shows an example of a simple PDF file represented in qpdf JSON format.

```
{
  "qpdf": [
    {
      "jsonversion": 2,
      "pdfversion": "1.3",
      "pushedinheritedpageresources": false,
      "calledgetallpages": false,
      "maxobjectid": 6
    },
    {
      "obj:1 0 R": {
        "value": {
          "/Pages": "3 0 R",
          "/Type": "/Catalog"
        }
      },
      "obj:2 0 R": {
        "value": {
          "/Author": "u:Digits of π",
          "/CreationDate": "u:D:20220731155308-05'00'",
          "/Creator": "u:A person typing in Emacs",
          "/Keywords": "u:potato, example",
          "/ModDate": "u:D:20220731155308-05'00'",
          "/Producer": "u:qpdf",
          "/Subject": "u:Example",
```

(continues on next page)

(continued from previous page)


```
      "/Title": "u:Something potato-related"
    }
  },
  "obj:3 0 R": {
    "value": {
      "/Count": 1,
      "/Kids": [
        "4 0 R"
      ],
      "/Type": "/Pages"
    }
  },
  "obj:4 0 R": {
    "value": {
      "/Contents": "5 0 R",
      "/MediaBox": [
        0,
        0,
        612,
        792
      ],
      "/Parent": "3 0 R",
      "/Resources": {
        "/Font": {
          "/F1": "6 0 R"
        }
      },
      "/Type": "/Page"
    }
  },
  "obj:5 0 R": {
    "stream": {
      "data": "eJxzCuFSUNB3M1QwMlEISQOyzY2AyEAhJAXI1gjIL0ksyddUCMnicg3hAgDLAQnI",
      "dict": {
        "/Filter": "/FlateDecode"
      }
    }
  },
  "obj:6 0 R": {
    "value": {
      "/BaseFont": "/Helvetica",
      "/Encoding": "/WinAnsiEncoding",
      "/Subtype": "/Type1",
      "/Type": "/Font"
    }
  },
  "trailer": {
    "value": {
      "/ID": [
        "b:98b5a26966fba4d3a769b715b2558da6",
        "b:6bea23330e0b9ff0ddb47b6757fb002e"
      ],
```


(continues on next page)

(continued from previous page)

```
          "/Info": "2 0 R",
          "/Root": "1 0 R",
          "/Size": 7
        }
      }
    }
  ]
}
```

10.4.3 qpdf JSON Input

The qpdf JSON output can be used in two different ways:

- By using the *--json-input* flag or calling `QPDF::createFromJSON` in place of `QPDF::processFile`, a qpdf JSON file can be used in place of a PDF file as the input to qpdf.
- By using the *--update-from-json* flag or calling `QPDF::updateFromJSON` on an initialized `QPDF` object, a qpdf JSON file can be used to apply changes to an existing `QPDF` object. That `QPDF` object can have come from any source including a PDF file, a qpdf JSON file, or the result of any other process that results in a valid, initialized `QPDF` object.

Here are some important things to know about qpdf JSON input.

- When a qpdf JSON file is used as the primary input file, it must be complete. This means
  - A JSON version number must be specified with the `"jsonversion"` key in the first array element
  - A PDF version number must be specified with the `"pdfversion"` key in the first array element
  - Stream data must be present for all streams
  - The trailer dictionary must be present, though only the `"/Root"` key is required.
- Certain fields from the input are ignored whether creating or updating from a JSON file:
  - `"maxobjectid"` is ignored, so it is not necessary to update it when adding new objects.
  - `"/Length"` is ignored in all stream dictionaries. qpdf doesn't put it there when it creates JSON output, and it is not necessary to add it.
  - `"/Size"` is ignored if it appears in a trailer dictionary as that is always recomputed by `QPDFWriter`.
  - Unknown keys at the top level of the file, within `"qpdf"`, and at the top level of each individual PDF object (inside the dictionary that has the `"value"` or `"stream"` key) and directly within `"stream"` are ignored for future compatibility. This includes other top-level keys generated by `qpdf` itself (such as `"pages"`). As such, those keys don't have to be consistent with the `"qpdf"` key if modifying a JSON file for conversion back to PDF. If you wish to store application-specific metadata, you can do so by adding a key whose name starts with `x-`. qpdf is guaranteed not to add any of its own keys that starts with `x-`. Note that any `"version"` key at the top level is ignored. The JSON version is obtained from the `"jsonversion"` key of the first element of the `"qpdf"` field.
- The values of `"calledgetallpages"` and `"pushedinheritedpageresources"` are ignored when creating a file. When updating a file, they treated as `false` if omitted.
- When qpdf reads a PDF file, the internal object numbers are always preserved. However, when qpdf writes a file using `QPDFWriter`, `QPDFWriter` does its own numbering and, in general, does not preserve input object numbers. That means that a qpdf JSON file that is used to update an existing PDF must have object numbers that match the input file it is modifying. In practical terms, this means that you can't use a JSON file created from one PDF file to modify the *output of running qpdf on that file*.

To put this more concretely, the following is valid:

```
qpdf --json-output in.pdf pdf.json
# edit pdf.json
qpdf in.pdf out.pdf --update-from-json=pdf.json
```

The following will produce unpredictable and probably incorrect results because `out.pdf` won't have the same object numbers as `pdf.json` and `in.pdf`.

```
qpdf --json-output in.pdf pdf.json
# edit pdf.json
qpdf in.pdf out.pdf --update-from-json=pdf.json
# edit pdf.json again
# Don't do this
qpdf out.pdf out2.pdf --update-from-json=pdf.json
```

- When updating from a JSON file (*--update-from-json*, `QPDF::updateFromJSON`), existing objects are updated in place. This has the following implications:
  - If the object you are updating is a stream, you may omit both `"data"` and `"datafile"`. In that case the original stream data is preserved. You must always provide a stream dictionary, but it may be empty. Note that an empty stream dictionary will clear the old dictionary. There is no way to indicate that an old stream dictionary should be left alone, so if your intention is to replace the stream data and preserve the dictionary, the original dictionary must appear in the JSON file.
  - You can change one object type to another object type including replacing a stream with a non-stream or a non-stream with a stream. If you replace a non-stream with a stream, you must provide data for the stream.
  - Objects that you do not wish to modify can be omitted from the JSON. That includes the trailer. That means you can use the output of a qpdf JSON file that was written using *--json-object* to have it include only the objects you intend to modify.
  - You can omit the `"pdfversion"` key. The input PDF version will be preserved.

10.4.4 qpdf JSON Workflow: CLI

This section includes a few examples of using qpdf JSON.

- Convert a PDF file to JSON format, edit the JSON, and convert back to PDF. This is an alternative to using QDF mode (see *QDF Mode*) to modify PDF files in a text editor. Each method has its own advantages and disadvantages.

```
qpdf --json-output in.pdf pdf.json
# edit pdf.json
qpdf --json-input pdf.json out.pdf
```

- Extract only a specific object into a JSON file, modify the object in JSON, and use the modified object to update the original PDF. In this case, we're editing object 4, whatever that may happen to be. You would have to know through some other means which object you wanted to edit, such as by looking at other JSON output or using a tool (possibly but not necessarily qpdf) to identify the object.

```
qpdf --json-output in.pdf pdf.json --json-object=4,0
# edit pdf.json
qpdf in.pdf --update-from-json=pdf.json out.pdf
```

Rather than using *--json-object* as in the above example, you could edit the JSON file to remove the objects you didn't need. You could also just leave them there, though the update process would be slower.

You could also add new objects to a file by adding them to `pdf.json`. Just be sure the object number doesn't conflict with an existing object. The `"maxobjectid"` field in the original output can help with this. You don't have to update it if you add objects as it is ignored when the file is read back in.

- Use *--json-input* and *--json-output* together to demonstrate preservation of object numbers. In this example, `a.json` and `b.json` will have the same objects and object numbers. The files may not be identical since strings may be normalized, fields may appear in a different order, etc. However `b.json` and `c.json` are probably identical.

```
qpdf --json-output in.pdf a.json
qpdf --json-input --json-output a.json b.json
qpdf --json-input --json-output b.json c.json
```

10.4.5 qpdf JSON Workflow: API

Everything that can be done using the qpdf CLI can be done using the C++ API. See comments in `QPDF.hh` for `writeJSON`, `createFromJSON`, and `updateFromJSON` for details.

10.5 JSON Compatibility 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 is a dictionary (JSON "object"). The JSON output contains various nested dictionaries and arrays. With the exception of dictionaries that are populated by the fields of PDF objects from the file, all instances of a dictionary are guaranteed to have exactly the same keys.

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. Note that, starting with version 2, the JSON version also appears in the `"jsonversion"` field of the first element of `"qpdf"` field.

Within a specific qpdf JSON version, 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. That means that consumers of qpdf JSON should ignore keys they don't know about.

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. It is possible for a key to be present and have a value that is explicitly `null`. 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 value as a 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 value including `null`.
- A single-element array in the help output indicates that the corresponding location in the actual output is either a single item or is an array of any length. The single item or each element of the array has whatever format is implied by the single element of the help output's array.
- A multi-element array in the help output indicates that the corresponding location in the actual output is an array of the same length. Each element of the output array has whatever format is implied by the corresponding 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 summary data. This is analogous to how qpdf's library interface works. The summary 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. It is especially useful to create a JSON file with the `"pages"` and `"qpdf"` keys and to use the `"pages"` information to find a page rather than navigating the pages tree manually. This can be done safely, and changes can made to the objects dictionary without worrying about keeping `"pages"` up to date since it is ignored when reading the file back in.

10.6 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:

- If a PDF file has certain types of errors in its pages tree (such as page objects that are direct or multiple pages sharing the same object ID), qpdf will automatically repair the pages tree. If you specify `"qpdf"` (or, with qpdf JSON version 1, `"objects"` or `"objectinfo"`) without any other keys, you will see the original pages tree without any corrections. If you specify any of keys that require page tree traversal (for example, `"pages"`, `"outlines"`, or `"pagelabel"`), then `"qpdf"` (and `"objects"` and `"objectinfo"`) will show the repaired page tree so that object references will be consistent throughout the file. You can tell if this has happened by looking at the `"calledgetallpages"` and `"pushedinheritedpageresources"` fields in the first element of the `"qpdf"` array.
- 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-key*, 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 in a real programming language than it is to add 1 in 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 that was 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`**.

- The `encrypt` key's values will be populated for non-encrypted files. Some values will be null, and others will have values that apply to unencrypted files.
- The qpdf library itself never loads an entire PDF into memory. This remains true for PDF files represented in JSON format. In general, qpdf will hold the entire object structure in memory once a file has been fully read (objects are loaded into memory lazily but stay there once loaded), but it will never have more than two copies of a stream in memory at once. That said, if you ask qpdf to write JSON to memory, it will do so, so be careful about this if you are working with very large PDF files. There is nothing in the qpdf library itself that prevents working with PDF files much larger than available system memory. qpdf can both read and write such files in JSON format. If you need to work with a PDF file's json representation in memory, it is recommended that you use either `none` or `file` as the argument to *`--json-stream-data`*, or if using the API, use `qpdf_sj_none` or `pdf_sj_file` as the json stream data value. If using `none`, you can use other means to obtain the stream data.

10.7 Changes from JSON v1 to v2

The following changes were made to qpdf's JSON output format for version 2.

- The representation of objects has changed. For details, see *qpdf JSON Object Representation*.
  - The representation of strings is now unambiguous for all strings. Strings a prefixed with either `u:` for Unicode strings or `b:` for byte strings.
  - Names are shown in qpdf's canonical form rather than in PDF syntax. (Example: the PDF-syntax name `/text#2fplain` appeared as `"/text#2fplain"` in v1 but appears as `"/text/plain"` in v2.
  - The top-level representation of an object in `"objects"` is a dictionary containing either a `"value"` key or a `"stream"` key, making it possible to distinguish streams from other objects.
- The `"objectinfo"` and `"objects"` keys have been removed in favor of a representation in `"qpdf"` that includes header information and differentiates between a stream and other kinds of objects. In v1, it was not possible to tell a stream from a dictionary within `"objects"`, and the PDF version was not captured at all.
- Within the objects dictionary, keys are now `"obj:`*`O G`*` R"` where *O* and *G* are the object and generation number. `"trailer"` remains the key for the trailer dictionary. In v1, the `obj:` prefix was not present. The rationale for this change is as follows:
  - Having a unique prefix (`obj:`) makes it much easier to search in the JSON file for the definition of an object
  - Having the key still contain `O G R` makes it much easier to construct the key from an indirect reference. You just have to prepend `obj:`. There is no need to parse the indirect object reference.
- In the `"encrypt"` object, the `"modifyannotations"` was misspelled as `"moddifyannotations"` in v1. This has been corrected.

10.7.1 Motivation for qpdf JSON version 2

qpdf JSON version 2 was created to make it possible to manipulate PDF files using JSON syntax instead of native PDF syntax. This makes it possible to make low-level updates to PDF files from just about any programming language or even to do so from the command-line using tools like `jq` or any editor that's capable of working with JSON files. There were several limitations of JSON format version 1 that made this impossible:

- Strings, names, and indirect object references in the original PDF file were all converted to strings in the JSON representation. For casual human inspection, this was fine, but in the general case, there was no way to tell the

difference between a string that looked like a name or indirect object reference from an actual name or indirect object reference.

- PDF strings were not unambiguously represented in the JSON format. The way qpdf JSON v1 represented a string was to try to convert the string to UTF-8. This was done by assuming a string that was not explicitly marked as Unicode was encoded in PDF doc encoding. The problem is that there is not a perfect bidirectional mapping between Unicode and PDF doc encoding, so if a binary string happened to contain characters that couldn't be bidirectionally mapped, there would be no way to get back to the original PDF string. Even when possible, trying to map from the JSON representation of a binary string back to the original string required knowledge of the mapping between PDF doc encoding and Unicode.
- There was no representation of stream data. If you wanted to extract stream data, you could use `--show-object`, so this wasn't that important for inspection, but it was a blocker for being able to go from JSON back to PDF. qpdf JSON version 2 allows stream data to be included inline as base64-encoded data. There is also an option to write all stream data to external files, which makes it possible to work with very large PDF files in JSON format even with tools that try to read the entire JSON structure into memory.
- The PDF version from PDF header was not represented in qpdf JSON v1.

CHAPTER

ELEVEN

CONTRIBUTING TO QPDF

11.1 Source Repository

The qpdf source code lives at https://github.com/qpdf/qpdf.

Create issues (bug reports, feature requests) at https://github.com/qpdf/qpdf/issues. If you have a general question or topic for discussion, you can create a discussion at https://github.com/qpdf/qpdf/discussions.

11.2 Code Formatting

The qpdf source code is formatting using clang-format $\geq$ version 15 with a `.clang-format` file at the top of the source tree. The `format-code` script reformats all the source code in the repository. You must have `clang-format` in your path, and it must be at least version 15.

For emacs users, the `.dir-locals.el` file configures emacs `cc-mode` for an indentation style that is similar to but not exactly like what `clang-format` produces. When there are differences, `clang-format` is authoritative. It is not possible to make `cc-mode` and `clang-format` exactly match since the syntax parser in emacs is not as sophisticated.

Blocks of code that should not be formatted can be surrounded by the comments `// clang-format off` and `// clang-format on`. Sometimes clang-format tries to combine lines in ways that are undesirable. In this case, we follow a convention of adding a comment `// line-break` on its own line.

For exact details, consult `.clang-format`. Here is a broad, partial summary of the formatting rules:

- Use spaces, not tabs.
- Keep lines to 80 columns when possible.
- Braces are on their own lines after classes and functions (and similar top-level constructs) and are compact otherwise.
- Closing parentheses are attached to the previous material, not not their own lines.

The `README-maintainer` file has a few additional notes that are probably not important to anyone who is not making deep changes to qpdf.

11.3 Automated Tests

The testing style of qpdf has evolved over time. More recent tests call `assert()`. Older tests print stuff to standard output and compare the output against reference files. Many tests are a mixture of these techniques.

The qtest style of testing is to test everything through the application. So effectively most testing is "integration testing" or "end-to-end testing".

For details about `qtest`, consult the QTest Manual. As you read it, keep in mind that, in spite of the recent date on the file, the vast majority of that documentation is from before 2007 and predates many test frameworks and approaches that are in use today.

Notes on testing:

- In most cases, things in the code are tested through integration tests, though the test suite is very thorough. Many tests are driven through the `qpdf` CLI. Others are driven through other files in the `qpdf` directory, especially `test_driver.cc` and `qpdf-ctest.c`. These programs only use the public API.
- In some cases, there are true "unit tests", but they are exercised through various stand-alone programs that exercise the library in particular ways, including some that have access to library internals. These are in the `libtests` directory.

11.3.1 Coverage

You wil see calls to `QTC::TC` throughout the code. This is a "manual coverage" system described in depth in the qtest documentation linked above. It works by ensuring that `QTC::TC` is called sometime during the test in each configured way. In brief:

- `QTC::TC` takes two mandatory options and an optional one:
  - The first two arguments must be *string literals*. This is because `qtest` finds coverage cases lexically.
  - The first argument is the scope name, usually `qpdf`. This means there is a `qpdf.testcov` file in the source directory.
  - The second argument is a case name. Each case name appears in `qpdf.testcov` with a number after it, usually `0`.
  - If the third argument is present, it is a number. `qtest` ensures that the `QTC::TC` is called for that scope and case at least once with the third argument set to every value from `0` to `n` inclusive, where `n` is the number after the coverage call.
- `QTC::TC` does nothing unless certain environment variables are set. Therefore, `QTC:TC` calls should have no side effects. (In some languages, they may be disabled at compile-time, though qpdf does not actually do this.)

So, for example, if you have this code:

```
QTC::TC("qpdf", "QPDF eof skipping spaces before xref",
        skipped_space ? 0 : 1);
```

and this line in *qpdf.testcov*:

```
QPDF eof skipping spaces before xref 1
```

the test suite will only pass if that line of code was called at least once with `skipped_space == 0` and at least once with `skipped_space == 1`.

The manual coverage approach ensures the reader that certain conditions were covered in testing. Use of `QTC::TC` is only part of the overall strategy.

I do not require testing on pull requests, but they are appreciated, and I will not merge any code that is not tested. Often someone will submit a pull request that is not adequately tested but is a good contribution. In those cases, I will often take the code, add it with tests, and accept the changes that way rather than merging the pull request as submitted.

11.4 Personal Comments

QPDF started as a work project in 2002. The first open source release was in 2008. While there have been a handful of contributors, the vast majority of the code was written by one person over many years as a side project.

I maintain a very strong commitment to backward compatibility. As such, there are many aspects of the code that are showing their age. While I believe the codebase to have high quality, there are things that I would do differently if I were doing them from scratch today. Sometimes people will suggest changes that I like but can't accept for backward compatibility reasons.

While I welcome contributions and am eager to collaborate with contributors, I have a high bar. I only accept things I'm willing to maintain over the long haul, and I am happy to help people get submissions into that state.

CHAPTER

TWELVE

DESIGN AND LIBRARY NOTES

12.1 Introduction

This section was written prior to the implementation of the qpdf library 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. That said, qpdf is not a strict PDF checker. There are many ways in which a PDF file can be out of conformance to the spec that qpdf doesn't notice or report.

12.2 Design Goals

The qpdf library includes support for reading and rewriting PDF files. It aims to hide from the user details involving object locations, modified (appended) PDF files, use of object streams, 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.

When accessing objects, a user of the library never has to care whether an object is direct or indirect as all access to objects deals with this transparently. All memory management details are also handled by the library. When modifying objects, it is possible to determine whether an object is indirect and to make copies of the object if needed.

Memory is managed mostly with `std::shared_ptr` object to minimize explicit memory handling. 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. The QPDFObjectHandle object contains an internal shared pointer to the underlying object. 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, object is initially *unresolved*. In this case, the first attempt to access the underlying object will result in the object 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. In cases where it is necessary to know whether an object is indirect or not, this information can be obtained from the QPDFObjectHandle. It is also possible to convert direct objects to indirect objects and vice versa.

Instances of QPDFObjectHandle can be directly created and modified using static factory methods in the QPDFObjectHandle class. There are factory methods for each type of object as well as a convenience method QPDFObjectHandle::parse that creates an object from a string representation of the object. The _qpdf user-defined string literal is also available, making it possible to create instances of QPDFObjectHandle with "(pdf-syntax)"_qpdf. 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 or with QPDF::create(). 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. 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 or 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 *Helper Classes*.

12.3 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 to 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 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.

12.4 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.

In a PDF file, objects may be direct or indirect. Direct objects are objects whose representations appear directly in PDF syntax. Indirect objects are references to objects by their ID. The qpdf library uses the QPDFObjectHandle type to hold onto objects and to abstract away in most cases whether the object is direct or indirect.

Internally, QPDFObjectHandle holds onto a shared pointer to the underlying object value. When a direct object is created programmatically by client code (rather than being read from the file), the QPDFObjectHandle that holds it is not associated with a QPDF object. When an indirect object reference is created, it starts off in an *unresolved* state and must be associated with a QPDF object, which is considered its *owner*. To access the actual value of the object, the object must be *resolved*. This happens automatically when the the object is accessed in any way.

To resolve an object, qpdf checks its object cache. If not found in the cache, it attempts to read the object from the input source associated with the QPDF object. If it is not found, a null object is returned. A null object is an object type, just like boolean, string, number, etc. It is not a null pointer. The PDF specification states that an indirect reference to an object that doesn't exist is to be treated as a null. The resulting object, whether a null or the actual object that was read, is stored in the cache. If the object is later replaced or swapped, the underlying object remains the same, but its value is replaced. This way, if you have a QPDFObjectHandle to an indirect object and the object by that number is replaced (by calling QPDF::replaceObject or QPDF::swapObjects), your QPDFObjectHandle will reflect the new value of the object. This is consistent with what would happen to PDF objects if you were to replace the definition of an object in the file.

When reading an object from the input source, 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 is 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 changes state and starts accumulating the keys and values of the dictionary.
- 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 obtain an indirect object handle from the QPDF class. The QPDF class consults its cache, and if necessary, inserts a new unresolved object, and returns an object handle pointing to the cache entry, which is then pushed onto the stack. When >> is finally read, the stack is converted into a QPDF_Dictionary (not directly accessible through the API) 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 the root object by getting the value of the /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 an unresolved indirect object, so it asks QPDF to resolve it. QPDF checks the cross reference table, gets the offset, and reads the object present at that offset. The object cache entry's unresolved value is replaced by the actual value, which causes any previously unresolved QPDFObjectHandle objects that pointed there to now have a shared copy of the actual object. Modifications through any such QPDFObjectHandle will be reflected in all of them. As the client continues to request objects, the same process is followed for each new requested object.

12.5 QPDF Object Internals

The internals of QPDFObjectHandle and how qpdf stores objects were significantly rewritten for QPDF 11. Here are some additional details.

12.5.1 Object Internals

The QPDF object has an object cache which contains a shared pointer to each object that was read from the file or added as an indirect object. Changes can be made to any of those objects through QPDFObjectHandle methods. Any such changes are visible to all QPDFObjectHandle instances that point to the same object. When a QPDF object is written by QPDFWriter or serialized to JSON, any changes are reflected.

12.5.2 Objects in qpdf 11 and Newer

The object cache in QPDF contains a shared pointer to QPDFObject. Any QPDFObjectHandle resolved from an indirect reference to that object has a copy of that shared pointer. Each QPDFObject object contains a shared pointer to an object of type QPDFValue. The QPDFValue type is an abstract base class. There is an implementation for each of the basic object types (array, dictionary, null, boolean, string, number, etc.) as well as a few special ones including uninitialized, unresolved, reserved, and destroyed. When an object is first created, its underlying QPDFValue has type unresolved. When the object is first accessed, the QPDFObject in the cache has its internal QPDFValue replaced with the object as read from the file. Since it is the QPDFObject object that is shared by all referencing QPDFObjectHandle objects as well as by the owning QPDF object, this ensures that any future changes to the object, including replacing the object with a completely different one by calling QPDF::replaceObject or QPDF::swapObjects, will be reflected across all QPDFObjectHandle objects that reference it.

A QPDFValue that originated from a PDF input source maintains a pointer to the QPDF object that read it (its *owner*). When that QPDF object is destroyed, it disconnects all objects reachable from it by clearing their owner. For indirect objects (all objects in the object cache), it also replaces the object's value with an object of type destroyed. This means that, if there are still any referencing QPDFObjectHandle objects floating around, requesting their owning QPDF will return a null pointer rather than a pointer to a QPDF object that is either invalid or points to something else, and any attempt to access an indirect object that is associated with a destroyed QPDF object will throw an exception. This operation also has the effect of breaking any circular references (which are common and, in some cases, required by the PDF specification), thus preventing memory leaks when QPDF objects are destroyed.

12.5.3 Objects prior to qpdf 11

Prior to qpdf 11, the functionality of the QPDFValue and QPDFObject classes were contained in a single QPDFObject class, which served the dual purpose of being the cache entry for QPDF and being the abstract base class for all the different PDF object types. The behavior was nearly the same, but there were some problems:

- While changes to a QPDFObjectHandle through mutation were visible across all referencing QPDFObjectHandle objects, *replacing* an object with QPDF::replaceObject or QPDF::swapObjects would leave QPDF with no way of notifying QPDFObjectHandle objects that pointed to the old QPDFObject. To work around this, every attempt to access the underlying object that a QPDFObjectHandle pointed to had to ask the owning QPDF whether the object had changed, and if so, it had to replace its internal QPDFObject pointer. This added overhead to every indirect object access even if no objects were ever changed.
- When a QPDF object was destroyed, any QPDFObjectHandle objects that referenced it would maintain a potentially invalid pointer as the owning QPDF. In practice, this wasn't usually a problem since generally people would have no need to maintain copies of a QPDFObjectHandle from a destroyed QPDF object, but in cases where this was possible, it was necessary for other software to do its own bookkeeping to ensure that an object's owner was still valid.

These problems were solved by splitting QPDFObject into QPDFObject and QPDFValue.

12.6 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 automatic 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.

12.7 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 effect 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. If you attempt to create an encrypted file that is not secure, qpdf will warn you and require you to explicitly state your intention to create an insecure file. So while qpdf can create insecure files, it won't let you do it by mistake.

12.8 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`.

12.9 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`.

12.10 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 *Copying Objects From Other PDF Files*. For an example of how to use reserved objects, search for newReserved in test_driver.cc in qpdf's sources.

12.11 Copying Objects From Other PDF Files

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

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

When you copy objects from one QPDF to another, the input source of the original file remain valid until you have finished with the destination object. This is because the input source is still used to retrieve any referenced stream data from the copied object. If needed, there are methods to force the data to be copied. See comments near the declaration of copyForeignObject in include/qpdf/QPDF.hh for details.

12.12 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 *Linearization* for a description of linearization is implemented. This section describes how we write non-linearized files including the creation of QDF files (see *QDF Mode*).

This outline was written prior to implementation and is not exactly accurate, but it portrays the essence 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 `:samp:`{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`.

12.13 Filtered Streams

Support for streams is implemented through the `Pipeline` interface which was designed for this library.

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.

12.14 Object Accessor Methods

For general information about how to access instances of `QPDFObjectHandle`, please see the comments in `QPDFObjectHandle.hh`. Search for "Accessor methods". This section provides a more in-depth discussion of the behavior and the rationale for the behavior.

Why were type errors made into warnings? When type checks were introduced into qpdf in the early days, it was expected that type errors would only occur as a result of programmer error. However, in practice, type errors would occur with malformed PDF files because of assumptions made in code, including code within the qpdf library and code written by library users. The most common case would be chaining calls to `getKey()` to access keys deep within a dictionary. In many cases, qpdf would be able to recover from these situations, but the old behavior often resulted in crashes rather than graceful recovery. For this reason, the errors were changed to warnings.

Why even warn about type errors when the user can't usually do anything about them? Type warnings are extremely valuable during development. Since it's impossible to catch at compile time things like typos in dictionary key names or logic errors around what the structure of a PDF file might be, the presence of type warnings can save lots of developer time. They have also proven useful in exposing issues in qpdf itself that would have otherwise gone undetected.

Can there be a type-safe `QPDFObjectHandle`? At the time of the release of qpdf 11, there is active work being done toward the goal of creating a way to work with PDF objects that is more type-safe and closer in feel to the current C++ standard library. It is hoped that this work will make it easier to write bindings to qpdf in modern languages like Rust. If this happens, it will likely be by providing an alternative to `QPDFObjectHandle` that provides a separate path to the underlying object. Details are still being worked out. Fundamentally, PDF objects are not strongly typed. They are similar to `JSON` objects or to objects in dynamic languages like Python: there are certain things you can only do to objects of a given type, but you can replace an object of one type with an object of another. Because of this, there will always be some checks that will happen at runtime.

Why does the behavior of a type exception differ between the C and C++ API? There is no way to throw and catch exceptions in C short of something like `setjmp` and `longjmp`, and that approach is not portable across language barriers. Since the C API is often used from other languages, it's important to keep things as simple as possible. Starting in qpdf 10.5, exceptions that used to crash code using the C API will be written to stderr by default, and it is possible to register an error handler. There's no reason that the error handler can't simulate exception handling in some way, such as by using `setjmp` and `longjmp` or by setting some variable that can be checked after library calls are made. In retrospect, it might have been better if the C API object handle methods returned error codes like the other methods and set return values in passed-in pointers, but this would complicate both the implementation and the use of the library for a case that is actually quite rare and largely avoidable.

How can I avoid type warnings altogether? For each `getSomethingValue` accessor that returns a value of the requested type and issues a warning for objects of the wrong type, there is also a `getValueAsSomething` method (since qpdf 10.6) that returns false for objects of the wrong type and otherwise returns true and initializes a reference. These

methods never generate type warnings and provide an alternative to explicitly checking the type of an object before calling an accessor method.

12.15 Smart Pointers

This section describes changes to the use of smart pointers that were made in qpdf 10.6.0 and 11.0.0.

In qpdf 11.0.0, `PointerHolder` was replaced with `std::shared_ptr` in qpdf's public API. A backward-compatible `PointerHolder` class has been provided that makes it possible for most code to remain unchanged. `PointerHolder` may eventually be removed from qpdf entirely, but this will not happen for a while to make it easier for people who need to support multiple versions of qpdf.

In 10.6.0, some enhancements were made to `PointerHolder` to ease the transition. These intermediate steps are relevant only for versions 10.6.0 through 10.6.3 but can still help with incremental modification of code.

The `POINTERHOLDER_TRANSITION` preprocessor symbol was introduced in qpdf 10.6.0 to help people transition from `PointerHolder` to `std::shared_ptr`. If you don't define this, you will get a compiler warning. Defining it to any value will suppress the warning. An explanation appears below of the different possible values for this symbol and what they mean.

Starting in qpdf 11.0.0, including `<qpdf/PointerHolder.hh>` defines the symbol `POINTERHOLDER_IS_SHARED_POINTER`. This can be used with conditional compilation to make it possible to support different versions of qpdf.

The rest of this section provides the details.

12.15.1 Transitional Enhancements to PointerHolder

In qpdf 10.6.0, some changes were to `PointerHolder` to make it easier to prepare for the transition to `std::shared_ptr`. These enhancements also make it easier to incrementally upgrade your code. The following changes were made to `PointerHolder` to make its behavior closer to that of `std::shared_ptr`:

- `get()` was added as an alternative to `getPointer()`
- `use_count()` was added as an alternative to `getRefcount()`
- A new global helper function `make_pointer_holder` behaves similarly to `std::make_shared`, so you can use `make_pointer_holder<T>(args...)` to create a `PointerHolder<T>` with `new T(args...)` as the pointer.
- A new global helper function `make_array_pointer_holder` takes a size and creates a `PointerHolder` to an array. It is a counterpart to the newly added `QUtil::make_shared_array` method, which does the same thing with a `std::shared_ptr`.

`PointerHolder` had a long-standing bug: a `const PointerHolder<T>` would only provide a `T const*` with its `getPointer` method. This is incorrect and is not how standard library C++ smart pointers or regular pointers behave. The correct semantics would be that a `const PointerHolder<T>` would not accept a new pointer after being created (`PointerHolder` has always behaved correctly in this way) but would still allow you to modify the item being pointed to. If you don't want to mutate the thing it points to, use `PointerHolder<T const>` instead. The new `get()` method behaves correctly. It is therefore not exactly the same as `getPointer()`, but it does behave the way `get()` behaves with `std::shared_ptr`. This shouldn't make any difference to any correctly written code.

12.15.2 Differences between PointerHolder and std::shared_ptr

Here is a list of things you need to think about when migrating from PointerHolder to std::shared_ptr. After the list, we will discuss how to address each one using the POINTERHOLDER_TRANSITION preprocessor symbol or other C++ coding techniques.

- PointerHolder<T> has an *implicit* constructor that takes a T*, which means you can assign a T* directly to a PointerHolder<T> or pass a T* to a function that expects a PointerHolder<T> as a parameter. std::shared_ptr<T> does not have this behavior, though you can still assign nullptr to a std::shared_ptr<T> and compare nullptr with a std::shared_ptr<T>. Here are some examples of how you might need to change your code:

  Old code:

  ```
  PointerHolder<X> x_p;
  X* x = new X();
  x_p = x;
  ```

  New code:

  ```
  auto x_p = std::make_shared<X>();
  X* x = x_p.get();
  // or, less safe, but closer:
  std::shared_ptr<X> x_p;
  X* x = new X();
  x_p = std::shared_ptr<X>(x);
  ```

  Old code:

  ```
  PointerHolder<Base> base_p;
  Derived* derived = new Derived();
  base_p = derived;
  ```

  New code:

  ```
  std::shared_ptr<Base> base_p;
  Derived* derived = new Derived();
  base_p = std::shared_ptr<Base>(derived);
  ```

- PointerHolder<T> has getPointer() to get the underlying pointer. It also has the seldom-used getRefcount() method to get the reference count. std::shared_ptr<T> has get() and use_count(). In qpdf 10.6, PointerHolder<T> also has get() and use_count().

12.15.3 Addressing the Differences

If you are not ready to take action yet, you can #define POINTERHOLDER_TRANSITION 0 before including any qpdf header file or add the definition of that symbol to your build. This will provide the backward-compatible PointerHolder API without any deprecation warnings. This should be a temporary measure as PointerHolder may disappear in the future. If you need to be able to support newer and older versions of qpdf, there are other options, explained below.

Note that, even with 0, you should rebuild and test your code. There may be compiler errors if you have containers of PointerHolder, but most code should compile without any changes. There are no uses of containers of PointerHolder in qpdf's API.

There are two significant things you can do to minimize the impact of switching from PointerHolder to std::shared_ptr:

- Use auto and decltype whenever possible when working with PointerHolder variables that are exchanged with the qpdf API.
- Use the POINTERHOLDER_TRANSITION preprocessor symbol to identify and resolve the differences described above.

To use POINTERHOLDER_TRANSITION, you will need to #define it before including any qpdf header files or specify its value as part of your build. The table below describes the values of POINTERHOLDER_TRANSITION. This information is also summarized in include/qpdf/PointerHolder.hh, so you will have it handy without consulting this manual.

Table 1: POINTERHOLDER_TRANSITION values

value	meaning
un-de-fined	Same as 0 but issues a warning
0	Provide a backward compatible PointerHolder and suppress all deprecation warnings; supports all prior qpdf versions
1	Make the PointerHolder<T>(T*) constructor explicit; resulting code supports all prior qpdf versions
2	Deprecate getPointer() and getRefcount(); requires qpdf 10.6.0 or later.
3	Deprecate all uses of PointerHolder; requires qpdf 11.0.0 or later
4	Disable all functionality from qpdf/PointerHolder.hh so that #include-ing it has no effect other than defining POINTERHOLDER_IS_SHARED_POINTER; requires qpdf 11.0.0 or later.

Based on the above, here is a procedure for preparing your code. This is the procedure that was used for the qpdf code itself.

You can do these steps without breaking support for qpdf versions before 10.6.0:

- Find all occurrences of PointerHolder in the code. See whether any of them can just be outright replaced with std::shared_ptr or std::unique_ptr. If you have been using qpdf prior to adopting C++11 and were using PointerHolder as a general-purpose smart pointer, you may have cases that can be replaced in this way.

  For example:

  - Simple PointerHolder<T> construction can be replaced with either the equivalent std::shared_ptr<T> construction or, if the constructor is public, with std::make_shared<T>(args...). If you are creating a smart pointer that is never copied, you may be able to use std::unique_ptr<T> instead.
  - Array allocations will have to be rewritten.

    Allocating a PointerHolder to an array looked like this:

    ```
    PointerHolder<X> p(true, new X[n]);
    ```

    To allocate a std::shared_ptr to an array:

    ```
    auto p = std::shared_ptr<X>(new X[n], std::default_delete<X[]>());
    // If you don't mind using QUtil, there's QUtil::make_shared_array<X>(n).
    // If you are using c++20, you can use std::make_shared<X[]>(n)
    // to get a std::shared_ptr<X[]> instead of a std::shared_ptr<X>.
    ```

    To allocate a std::unique_ptr to an array:

```
auto p = std::make_unique<X[]>(n);
// or, if X has a private constructor:
auto p = std::unique_ptr<X[]>(new X[n]);
```

- If a PointerHolder<T> can't be replaced with a standard library smart pointer because it is used with an older qpdf API call, perhaps it can be declared using auto or decltype so that, when building with a newer qpdf API changes, your code will just need to be recompiled.
- #define POINTERHOLDER_TRANSITION 1 to enable deprecation warnings for all implicit constructions of PointerHolder<T> from a plain T*. When you find one, explicitly construct the PointerHolder<T>.
  - Old code:

```
PointerHolder<X> x = new X();
```

  - New code:

```
auto x = PointerHolder<X>(new X(...)); // all versions of qpdf
// or, if X(...) is public:
auto x = make_pointer_holder<X>(...); // only 10.6 and above
```

    Other examples appear above.

If you need to support older versions of qpdf than 10.6, this is as far as you can go without conditional compilation.

Starting in qpdf 11.0.0, including <qpdf/PointerHolder.hh> defines the symbol POINTERHOLDER_IS_SHARED_POINTER. If you want to support older versions of qpdf and still transition so that the backward-compatible PointerHolder is not in use, you can separate old code and new code by testing with the POINTERHOLDER_IS_SHARED_POINTER preprocessor symbol, as in

```
#include <qpdf/PointerHolder.hh>
#ifdef POINTERHOLDER_IS_SHARED_POINTER
std::shared_ptr<X> x;
#else
PointerHolder<X> x;
#endif // POINTERHOLDER_IS_SHARED_POINTER
x = decltype(x)(new X())
```

or

```
#include <qpdf/PointerHolder.hh>
#ifdef POINTERHOLDER_IS_SHARED_POINTER
auto x_p = std::make_shared<X>();
X* x = x_p.get();
#else
auto x_p = PointerHolder<X>(new X());
X* x = x_p.getPointer();
#endif // POINTERHOLDER_IS_SHARED_POINTER
x_p->doSomething();
x->doSomethingElse();
```

If you don't need to support older versions of qpdf, you can proceed with these steps without protecting changes with the preprocessor symbol. Here are the remaining changes.

- #define POINTERHOLDER_TRANSITION 2 to enable deprecation of getPointer() and getRefcount()

- Replace `getPointer()` with `get()` and `getRefcount()` with `use_count()`. These methods were not present prior to 10.6.0.

When you have gotten your code to compile cleanly with `POINTERHOLDER_TRANSITION=2`, you are well on your way to being ready for eliminating `PointerHolder` entirely. The code at this point will not work with any qpdf version prior to 10.6.0.

To support qpdf 11.0.0 and newer and remove `PointerHolder` from your code, continue with the following steps:

- Replace all occurrences of `PointerHolder` with `std::shared_ptr` except in the literal statement `#include <qpdf/PointerHolder.hh>`
- Replace all occurrences of `make_pointer_holder` with `std::make_shared`
- Replace all occurrences of `make_array_pointer_holder` with `QUtil::make_shared_array`. You will need to include `<qpdf/QUtil.hh>` if you haven't already done so.
- Make sure `<memory>` is included wherever you were including `<qpdf/PointerHolder.hh>`.
- If you were using any array `PointerHolder<T>` objects, replace them as above. You can let the compiler find these for you.
- `#define POINTERHOLDER_TRANSITION 3` to enable deprecation of all `PointerHolder<T>` construction.
- Build and test. Fix any remaining issues.
- If not supporting older versions of qpdf, remove all references to `<qpdf/PointerHolder.hh>`. Otherwise, you will still need to include it but can `#define POINTERHOLDER_TRANSITION 4` to prevent `PointerHolder` from being defined. The `POINTERHOLDER_IS_SHARED_POINTER` symbol will still be defined.

12.15.4 Historical Background

Since its inception, the qpdf library used its own smart pointer class, `PointerHolder`. The `PointerHolder` class was originally created long before `std::shared_ptr` existed, and qpdf itself didn't start requiring a C++11 compiler until version 9.1.0 released in late 2019. With current C++ versions, it is no longer desirable for qpdf to have its own smart pointer class.

CHAPTER

THIRTEEN

QPDFJOB: A JOB-BASED INTERFACE

All of the functionality from the **qpdf** command-line executable is available from inside the C++ library using the `QPDFJob` class. There are several ways to access this functionality:

- Command-line options
  - Run the **qpdf** command line
  - Use from the C++ API with `QPDFJob::initializeFromArgv`
  - Use from the C API with `qpdfjob_run_from_argv` from `qpdfjob-c.h`. If you are calling from a Windows-style main and have an argv array of `wchar_t`, you can use `qpdfjob_run_from_wide_argv`.
- The job JSON file format
  - Use from the CLI with the *`--job-json-file`* parameter
  - Use from the C++ API with `QPDFJob::initializeFromJson`
  - Use from the C API with `qpdfjob_run_from_json` from `qpdfjob-c.h`
  - Note: this is unrelated to *`--json`* but can be combined with it. For more information on qpdf JSON (vs. QPDFJob JSON), see *qpdf JSON*.
- The `QPDFJob` C++ API

If you can understand how to use the **qpdf** CLI, you can understand the `QPDFJob` class and the JSON file. qpdf guarantees that all of the above methods are in sync. Here's how it works:

Table 1: QPDFJob Interfaces

CLI	JSON	C++
`--some-option`	`"someOption": ""`	`config()->someOption()`
`--some-option=value`	`"someOption": "value"`	`config()->someOption("value")`
positional argument	`"otherOption": "value"`	`config()->otherOption("value")`

In the JSON file, the JSON structure is an object (dictionary) whose keys are command-line flags converted to camelCase. Positional arguments have some corresponding key, which you can find by running `qpdf` with the *`--job-json-help`* flag. For example, input and output files are named by positional arguments on the CLI. In the JSON, they appear in the `"inputFile"` and `"outputFile"` keys. The following are equivalent:

CLI:

```
qpdf infile.pdf outfile.pdf \
   --pages . other.pdf --password=x 1-5 -- \
   --encrypt user owner 256 --print=low -- \
   --object-streams=generate
```

Job JSON:

```
{
  "inputFile": "infile.pdf",
  "outputFile": "outfile.pdf",
  "pages": [
    {
      "file": "."
    },
    {
      "file": "other.pdf",
      "password": "x",
      "range": "1-5"
    }
  ],
  "encrypt": {
    "userPassword": "user",
    "ownerPassword": "owner",
    "256bit": {
      "print": "low"
    }
  },
  "objectStreams": "generate"
}
```

C++ code:

```
#include <qpdf/QPDFJob.hh>
#include <qpdf/QPDFUsage.hh>
#include <iostream>

int main(int argc, char* argv[])
{
    try
    {
        QPDFJob j;
        j.config()
            ->inputFile("infile.pdf")
            ->outputFile("outfile.pdf")
            ->pages()
            ->pageSpec(".", "1-z")
            ->pageSpec("other.pdf", "1-5", "x")
            ->endPages()
            ->encrypt(256, "user", "owner")
            ->print("low")
            ->endEncrypt()
            ->objectStreams("generate")
            ->checkConfiguration();
        j.run();
    }
    catch (QPDFUsage& e)
    {
        std::cerr << "configuration error: " << e.what() << std::endl;
        return 2;
```

(continues on next page)

(continued from previous page)

```
    }
    catch (std::exception& e)
    {
        std::cerr << "other error: " << e.what() << std::endl;
        return 2;
    }
    return 0;
}
```

Note the `QPDFUsage` exception above. This is thrown whenever a configuration error occurs. These exactly correspond to usage messages issued by the **qpdf** CLI for things like omitting an output file, specifying *–pages* multiple times, or other invalid combinations of options. `QPDFUsage` is thrown by the argv and JSON interfaces as well as the native `QPDFJob` interface.

It is also possible to mix and match command-line options and JSON from the CLI. For example, you could create a file called `my-options.json` containing the following:

```
{
  "encrypt": {
    "userPassword": "",
    "ownerPassword": "owner",
    "256bit": {
    }
  },
  "objectStreams": "generate"
}
```

and use it with other options to create 256-bit encrypted (but unrestricted) files with object streams while specifying other parameters on the command line, such as

```
qpdf infile.pdf outfile.pdf --job-json-file=my-options.json
```

See also `examples/qpdf-job.cc` in the source distribution as well as comments in `QPDFJob.hh`.

13.1 QPDFJob Design

This section describes some of the design rationale and history behind `QPDFJob`.

Documentation of `QPDFJob` is divided among three places:

- "HOW TO ADD A COMMAND-LINE ARGUMENT" in `README-maintainer` provides a quick reminder of how to add a command-line argument.
- The source file `generate_auto_job` has a detailed explanation about how `QPDFJob` and `generate_auto_job` work together.
- This chapter of the manual has other details.

Prior to qpdf version 10.6.0, the qpdf CLI executable had a lot of functionality built into it that was not callable from the library as such. This created a number of problems:

- Some of the logic in `qpdf.cc` was pretty complex, such as image optimization, generating JSON output, and many of the page manipulations. While those things could all be coded using the C++ API, there would be a lot of duplicated code.

- Page splitting and merging will get more complicated over time as qpdf supports a wider range of document-level options. It would be nice to be able to expose this to library users instead of baking it all into the CLI.
- Users of other languages who just wanted an interface to do things that the CLI could do didn't have a good way to do it, such as just handing a library call a set of command-line options or an equivalent JSON object that could be passed in as a string.
- The qpdf CLI itself was almost 8,000 lines of code. It needed to be refactored, cleaned up, and split.
- Exposing a new feature via the command-line required making lots of small edits to lots of small bits of code, and it was easy to forget something. Adding a code generator, while complex in some ways, greatly reduces the chances of error when extending qpdf.

Here are a few notes on some design decisions about QPDFJob and its various interfaces.

- Bare command-line options (flags with no parameter) map to config functions that take no options and to JSON keys whose values are required to be the empty string. The rationale is that we can later change these bare options to options that take an optional parameter without breaking backward compatibility in the CLI or the JSON. Options that take optional parameters generate two config functions: one has no arguments, and one that has a `char const*` argument. This means that adding an optional parameter to a previously bare option also doesn't break binary compatibility.
- Adding a new argument to `job.yml` automatically triggers almost everything by declaring and referencing things that you have to implement. This way, once you get the code to compile and link, you know you haven't forgotten anything. There are two tricky cases:
  - If an argument handler has to do something special, like call a nested config method or select an option table, you have to implement it manually. This is discussed in `generate_auto_job`.
  - When you add an option that has optional parameters or choices, both of the handlers described above are declared, but only the one that takes an argument is referenced. You have to remember to implement the one that doesn't take an argument or else people will get a linker error if they try to call it. The assumption is that things with optional parameters started out as bare, so the argument-less version is already there.
- If you have to add a new option that requires its own option table, you will have to do some extra work including adding a new nested Config class, adding a config member variable to `ArgParser` in `QPDFJob_argv.cc` and `Handlers` in `QPDFJob_json.cc`, and make sure that manually implemented handlers are consistent with each other. It is best to add explicit test cases for all the various ways to get to the option.

CHAPTER

FOURTEEN

LINEARIZATION

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

14.1 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.

14.2 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 *Optimization*. 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

14.3 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.

14.4 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.

14.5 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.)

14.6 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.

14.7 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

FIFTEEN

OBJECT AND CROSS-REFERENCE STREAMS

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

15.1 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 *Implications for Linearized Files* for details.

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

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

The object stream dictionary has the following keys:

- `/N`: number of objects
- `/First`: byte offset of first object
- `/Extends`: indirect reference to stream that this extends

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

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

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

15.2 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 `[:samp:`{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.

15.2.1 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.

15.3 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.

15.4 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.

CHAPTER

SIXTEEN

PDF ENCRYPTION

This chapter discusses PDF encryption in a general way with an angle toward how it works in **qpdf**. This chapter is not intended to replace the PDF specification. Please consult the spec for full details.

16.1 PDF Encryption Concepts

Encryption
: Encryption is the replacement of *clear text* with encrypted text, also known as *ciphertext*. The clear text may be retrieved from the ciphertext if the encryption key is known.

  PDF files consist of an object structure. PDF objects may be of a variety of types including (among others) numbers, boolean values, names, arrays, dictionaries, strings, and streams. In a PDF file, only strings and streams are encrypted.

Security Handler
: Since the inception of PDF, there have been several modifications to the way files are encrypted. Encryption is handled by a *security handler*. The *standard security handler* is password-based. This is the only security handler implemented by qpdf, and this material is all focused on the standard security handler. There are various flags that control the specific details of encryption with the standard security handler. These are discussed below.

Encryption Key
: This refers to the actual key used by the encryption and decryption algorithms. It is distinct from the password. The main encryption key is generated at random and stored encrypted in the PDF file. The passwords used to protect a PDF file, if any, are used to protect the encryption key. This design makes it possible to use different passwords (e.g., user and owner passwords) to retrieve the encryption key or even to change the password on a file without changing the encryption key. qpdf can expose the encryption key when run with the *--show-encryption-key* option and can accept a hex-encoded encryption key in place of a password when run with the *--password-is-hex-key* option.

Password Protection
: Password protection is distinct from encryption. This point is often misunderstood. A PDF file can be encrypted without being password-protected. The intent of PDF encryption was that there would be two passwords: a *user password* and an *owner password*. Either password can be used to retrieve the encryption key. A conforming reader is supposed to obey the security restrictions if the file is opened using the user password but not if the file is opened with the owner password. **qpdf** makes no distinction between which password is used to open the file. The distinction made by conforming readers between the user and owner password is what makes it common to create encrypted files with no password protection. This is done by using the empty string as the user password and some secret string as the owner password. When a user opens the PDF file, the empty string is used to retrieve the encryption key, making the file usable, but a conforming reader restricts certain operations from the user.

What does all this mean? Here are a few things to realize.

- Since the user password and the owner password are both used to recover the single encryption key, there is *fundamentally no way* to prevent an application from disregarding the security restrictions on a file. Any software that can read the encrypted file at all has the encryption key. Therefore, the security of the restrictions placed on PDF files is solely enforced by the software. Any open source PDF reader could be trivially modified to ignore the security restrictions on a file. The PDF specification is clear about this point. This means that PDF restrictions on non-password-protected files only restrict users who don't know how to circumvent them.
- If a file is password-protected, you have to know at least one of the user or owner password to retrieve the encryption key. However, in the case of 40-bit encryption, the actual encryption key is only 5 bytes long and can be easily brute-forced. As such, files encrypted with 40-bit encryption are not secure regardless of how strong the password is. With 128-bit encryption, the default security handler uses RC4 encryption, which is also known to be insecure. As such, the only way to securely encrypt a PDF file using the standard security handler (as of the last review of this chapter in 2022) is to use AES encryption. This is the only supported algorithm with 256-bit encryption, and it can be selected to be used with 128-bit encryption as well. However there is no reason to use 128-bit encryption with AES. If you are going to use AES, just use 256-bit encryption instead. The security of a 256-bit AES-encrypted PDF file with a strong password is comparable to using a general-purpose encryption tool like **`gpg`** or **`openssl`** to encrypt the PDF file with the same password, but the advantage of using PDF encryption is that no software is required beyond a regular PDF viewer.

16.2 PDF Encryption Details

This section describes a few details about PDF encryption. It does not describe all the details. For that, read the PDF specification. The details presented here, however, should go a long way toward helping a casual user/developer understand what's going on with encrypted PDF files.

Here are more concepts to understand.

Algorithm parameters `V` and `R`

There are two parameters that control the details of encryption using the standard security handler: `V` and `R`.

`V` is a code specifying the algorithms that are used for encrypting the file, handling keys, etc. It may have any of the following values:

Table 1: Encryption Algorithms: `V`

V	Meaning
1	The original algorithm, which encrypted files using 40-bit keys.
2	An extension of the original algorithm allowing longer keys. Introduced in PDF 1.4.
3	An unpublished algorithm that permits file encryption key lengths ranging from 40 to 128 bits. Introduced in PDF 1.4. qpdf is believed to be able to read files with `V` = 3 but does not write such files.
4	An extension of the algorithm that allows it to be parameterized by additional rules for handling strings and streams. Introduced in PDF 1.5.
5	An algorithm that allows specification of separate security handlers for strings and streams as well as embedded files, and which supports 256-bit keys. Introduced in PDF 1.7 extension level 3 and later extended in extension level 8. This is the encryption system in the PDF 2.0 specification, ISO-32000.

`R` is a code specifying the revision of the standard handler. It is tightly coupled with the value of `V`. `R` may have any of the following values:

Table 2: Relationship between R and V

R	Expected V
2	V must be 1
3	V must be 2 or 3
4	V must be 4
5	V must be 5; this extension was never fully specified and existed for a short time in some versions of Acrobat. **qpdf** is able to read and write this format, but it should not be used for any purpose other than testing compatibility with the format.
6	V must be 5. This is the only value that is not deprecated in the PDF 2.0 specification, ISO-32000.

Encryption Dictionary

Encrypted PDF files have an encryption dictionary. There are several fields, but these are the important ones for our purposes:

- V and R as described above
- O, U, OE, UE: values used by the algorithms that recover the encryption key from the user and owner password. Which of these are defined and how they are used vary based on the value of R.
- P: a bit field that describes which restrictions are in place. This is discussed below in *PDF Security Restrictions*

Encryption Algorithms

PDF files may be encrypted with the obsolete, insecure RC4 algorithm or the more secure AES algorithm. See also *Weak Cryptography* for a discussion. 40-bit encryption always uses RC4. 128-bit can use either RC4 (the default for compatibility reasons) or, starting with PDF 1.6, AES. 256-bit encryption always uses AES.

16.3 PDF Security Restrictions

PDF security restrictions are described by a bit field whose value is stored in the P field in the encryption dictionary. The value of P is used by the algorithms to recover the encryption key given the password, which makes the value of P tamper-resistent.

P is a 32-bit integer, treated as a signed twos-complement number. A 1 in any bit position means the permission is granted. The PDF specification numbers the bits from 1 (least significant bit) to 32 (most significant bit) rather than the more customary 0 to 31. For consistency with the spec, the remainder of this section uses the 1-based numbering.

Only bits 3, 4, 5, 6, 9, 10, 11, and 12 are used. All other bits are set to 1. Since bit 32 is always set to 1, the value of P is always a negative number. (**qpdf** recognizes a positive number on behalf of buggy writers that treat P as unsigned. Such files have been seen in the wild.)

Here are the meanings of the bit positions. All bits not listed must have the value 1 except bits 1 and 2, which must have the value 0. However, the values of bits other than those in the table are ignored, so having incorrect values probably doesn't break anything in most cases. A value of 1 indicates that the permission is granted.

Table 3: P Bit Values

Bit	Meaning
3	for R = 2 printing; for R $\geq$ 3, printing at low resolution
4	modifying the document except as controlled by bits 6, 9, and 11
5	extracting text and graphics for purposes other than accessibility to visually impaired users
6	add or modify annotations, fill in interactive form fields; if bit 4 is also set, create or modify interactive form fields
9	for R $\geq$ 3, fill in interactive form fields even if bit 6 is clear
10	not used; formerly granted permission to extract material for accessibility, but the specification now disallows restriction of accessibility, and conforming readers are to treat this bit as if it is set regardless of its value
11	for R $\geq$ 3, assemble document including inserting, rotating, or deleting pages or creating document outlines or thumbnail images
12	for R $\geq$ 3, allow printing at full resolution

16.4 How qpdf handles security restrictions

The section describes exactly what the qpdf library does with regard to P based on the various settings of different security options.

- Start with all bits set except bits 1 and 2, which are cleared
- Clear bits and described in the table below:

Table 4: Command-line Arguments and P Bit Values

R	Argument	Bits Cleared
R = 2	`--print=n`	3
R = 2	`--modify=n`	4
R = 2	`--extract=n`	5
R = 2	`--annotate=n`	6
R = 3	`--accessibility=n`	10
R $\geq$ 4	`--accessibility=n`	ignored
R $\geq$ 3	`--extract=n`	5
R $\geq$ 3	`--print=none`	3, 12
R $\geq$ 3	`--print=low`	12
R $\geq$ 3	`--modify=none`	4, 6, 9, 11
R $\geq$ 3	`--modify=assembly`	4, 6, 9
R $\geq$ 3	`--modify=form`	4, 6
R $\geq$ 3	`--modify=annotate`	4
R $\geq$ 3	`--assemble=n`	11
R $\geq$ 3	`--annotate=n`	6
R $\geq$ 3	`--form=n`	9
R $\geq$ 3	`--modify-other=n`	4

Options to **`qpdf`**, both at the CLI and library level, allow more granular clearing of permission bits than do most tools, including Adobe Acrobat. As such, PDF viewers may respond in surprising ways based on options passed to qpdf. If you observe this, it is probably not because of a bug in qpdf.

16.5 User and Owner Passwords

When you use qpdf to show encryption parameters and you open a file with the owner password, sometimes qpdf reveals the user password, and sometimes it doesn't. Here's why.

For `V` < 5, the user password is actually stored in the PDF file encrypted with a key that is derived from the owner password, and the main encryption key is encrypted using a key derived from the user password. When you open a PDF file, the reader first tries to treat the given password as the user password, using it to recover the encryption key. If that works, you're in with restrictions (assuming the reader chooses to enforce them). If it doesn't work, then the reader treats the password as the owner password, using it to recover the user password, and then uses the user password to retrieve the encryption key. This is why creating a file with the same user password and owner password with `V` < 5 results in a file that some readers will never allow you to open as the owner. When an empty owner password is given at file creation, the user password is used as both the user and owner password. Typically when a reader encounters a file with `V` < 5, it will first attempt to treat the empty string as a user password. If that works, the file is encrypted but not password-protected. If it doesn't work, then a password prompt is given.

For `V` $\geq$ 5, the main encryption key is independently encrypted using the user password and the owner password. There is no way to recover the user password from the owner password. Restrictions are imposed or not depending on which password was used. In this case, the password supplied, if any, is tried both as the user password and the owner password, and whichever works is used. Typically the password is tried as the owner password first. (This is what the PDF specification says to do.) As such, specifying a user password and leaving the owner password blank results in a file that is opened as owner with no password, effectively rendering the security restrictions useless. This is why **qpdf** requires you to pass *--allow-insecure* to create a file with an empty owner password when 256-bit encryption is in use.

CHAPTER

SEVENTEEN

RELEASE NOTES

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

11.2.0: November 20, 2022

- Build changes
  - A C++-17 compiler is now required.
- Library enhancements
  - Move stream creation functions in the `QPDF` object where they belong. The ones in `QPDFObjectHandle` are not deprecated and will stick around.
  - Add some convenience methods to `QPDFTokenizer::Token` for testing token types. This is part of qpdf's lexical layer and will not be needed by most developers.
- Bug fixes
  - Fix issue with missing symbols in the mingw build.
  - Fix major performance bug with the OpenSSL crypto provider. This bug was causing a 6x to 12x slowdown for encrypted files when OpenSSL 3 was in use. This includes the default Windows builds distributed with the qpdf release.
  - Fix obscure bug involving appended files that reuse an object number that was used as a cross reference stream in an earlier stage of the file.

11.1.1: October 1, 2022

- Bug fixes
  - Fix edge case with character encoding for strings whose initial characters happen to coincide with Unicode markers.
  - Fix issue with AppImage discarding the first command-line argument when invoked as the name of one of the embedded executables. Also, fix-qdf, for unknown reasons, had the wrong runpath and would use a qpdf library that was installed on the system.
- Test improvements
  - Exercise the case of `char` being `unsigned` by default in automated tests.
  - Add AppImage-specific tests to CI to ensure that the AppImage works in the various ways it is intended to be invoked.
- Other changes
  - Include more code tidying and performance improvements from M. Holger.

11.1.0: September 14, 2022

- Build fixes
  - Remove `LL_FMT` tests, which were broken for cross compilation. The code just uses `%lld` now.
  - Some symbols were not properly exported for the Windows DLL build.
  - Force project-specific header files to precede all others in the build so that a previous qpdf installation won't break building qpdf from source.
- Packaging note omitted from 11.0.0 release notes:
  - On GitHub, the release tags are now `vX.Y.Z` instead of `release-qpdf-X.Y.Z` to be more consistent with current practice.

11.0.0: September 10, 2022

- Replacement of `PointerHolder` with `std::shared_ptr`
  - The qpdf-specific `PointerHolder` smart pointer implementation has now been completely replaced with `std::shared_ptr` through the qpdf API. Please see *Smart Pointers* for details about this change and a comprehensive migration plan. Note that a backward-compatible `PointerHolder` class is provided and is enabled by default. A warning is issued, but this can be turned off by following the migration steps outlined in the manual.
- qpdf JSON version 2
  - qpdf's JSON output mode is now at version 2. This fixes several flaws with version 1. Version 2 JSON output is unambiguous and complete, and bidirectional conversion between JSON and PDF is supported. Command-line options and library API are available for creating JSON from PDF, creating PDF from JSON and updating existing PDF at the object level from JSON.
  - New command-line arguments: *--json-output*, *--json-input*, *--update-from-json*
  - New C++ API calls: `QPDF::writeJSON`, `QPDF::createFromJSON`, `QPDF::updateFromJSON`
  - New C API calls: `qpdf_create_from_json_file`, `qpdf_create_from_json_data`, `qpdf_update_from_json_file`, `qpdf_update_from_json_data`, and `qpdf_write_json`.
  - Complete documentation can be found at *qpdf JSON*. A comprehensive list of changes from version 1 to version 2 can be found at *Changes from JSON v1 to v2*.
- Build replaced with cmake
  - The old autoconf-based build has been replaced with CMake. CMake version 3.16 or newer is required. For details, please read *Building and Installing QPDF* and, if you package qpdf for a distribution, *Notes for Packagers*.
  - For the most part, other than being familiar with generally how to build things with cmake, what you need to know to convert your build over is described in *Converting From autoconf to cmake*. Here are a few changes in behavior to be aware of:
    * Example sources are installed by default in the documentation directory.
    * The configure options to enable image comparison and large file tests have been replaced by environment variables. The old options set environment variables behind the scenes. Before, to skip image tests, you had to set `QPDF_SKIP_TEST_COMPARE_IMAGES=1`, which was done by default. Now these are off by default, and you have to set `QPDF_TEST_COMPARE_IMAGES=1` to enable them.
    * In the default configuration, the native crypto provider is only selected when explicitly requested or when there are no other options. See *Build-time Crypto Selection* for a detailed discussion.
    * Windows external libraries are detected by default if the `external-libraries` directory is found. Static libraries for zlib, libjpeg, and openssl are provided as described in `README-windows.md`. They are only compatible with non-debug builds.

    * A new directory called `pkg-tests` has been added which contains short shell scripts that can be used to smoke test an installed qpdf package. These are used by the debian `autopkgtest` framework but can be used by others. See `pkg-test/README.md` for details.

- Performance improvements
  - Many performance enhancements have been added. In developer performance benchmarks, gains on the order of 20% have been observed. Most of that work, including major optimization of qpdf's lexical and parsing layers, was done by M. Holger.
- CLI: breaking changes
  - The *--show-encryption* flag now provides encryption information even if a correct password is not supplied. If you were relying on its not working in this case, see *--requires-password* for a reliable test.
  - The default json output version when *--json* is specified has been changed from `1` to `latest`, which is now `2`.
  - The *--allow-weak-crypto* flag is now mandatory when explicitly creating files with weak cryptographic algorithms. See *Weak Cryptography* for a discussion.
- API: breaking changes
  - Deprecate `QPDFObject.hh` for removal in qpdf 12. The only use case for including `qpdf/QPDFObject.hh` was to get `QPDFObject::object_type_e`. Since 10.5.0, this has been an alias to `qpdf_object_type_e`, defined in `qpdf/Constants.h`. To fix your code, replace any includes of `qpdf/QPDFObject.hh` with `qpdf/Constants.h`, and replace all occurrences of `QPDFObject::ot_` with `::ot_`. If you need your code to be backward compatible to qpdf versions prior to 10.5.0, you can check that the preprocessor symbol `QPDF_MAJOR_VERSION` is defined and `>= 11`. As a stop-gap, you can `#define QPDF_OBJECT_NOWARN` to suppress the warning.
  - `Pipeline::write` now takes `unsigned char const*` instead of `unsigned char*`. Callers don't need to change anything, but you no longer have to pass writable pointers to pipelines. If you've implemented your own pipeline classes, you will need to update them.
  - Remove deprecated `QPDFAcroFormDocumentHelper::copyFieldsFromForeignPage`. This method never worked and only did something in qpdf version 10.2.x.
  - Remove deprecated `QPDFNameTreeObjectHelper` and `QPDFNumberTreeObjectHelper` constructors that don't take a `QPDF&` argument.
  - The function passed to and called by `QPDFJob::doIfVerbose` now takes a `Pipeline&` argument instead of a `std::ostream&` argument.
  - Intentionally break API to call attention to operations that write files with insecure encryption:
    * Remove pre qpdf-8.4.0 encryption API methods from `QPDFWriter` and their corresponding C API functions
    * Add `Insecure` to the names of some `QPDFWriter` methods and `_insecure` to the names of some C API functions without otherwise changing their behavior
    * See *API-Breaking Changes in qpdf 11.0* for specific details, and see *Weak Cryptography* for a general discussion.
  - `QPDFObjectHandle::warnIfPossible` no longer takes an optional argument to throw an exception if there is no description. If there is no description, it writes to the default `QPDFLogger`'s error stream. (`QPDFLogger` is new in qpdf 11—see below.)
  - `QPDF` objects can no longer be copied or assigned to. It has never been safe to do this because of assumptions made by library code. Now it is prevented by the API. If you run into trouble, use

QPDF::create() to create QPDF shared pointers (or create them in some other way if you need backward compatibility with older qpdf versions).

- CLI Enhancements
  - qpdf --list-attachments --verbose includes some additional information about attachments. Additional information about attachments is also included in the attachments JSON key with --json.
  - For encrypted files, qpdf --json reveals the user password when the specified password did not match the user password and the owner password was used to recover the user password. The user password is not recoverable from the owner password when 256-bit keys are in use.
  - --verbose and --progress may be now used when writing the output PDF to standard output. In that case, the verbose and progress messages are written to standard error.
- Library Enhancements
  - A new object QPDFLogger has been added. Details are in include/qpdf/QPDFLogger.hh.
    * QPDF and QPDFJob both use the default logger by default but can have their loggers overridden. The setOutputStreams method is deprecated in both classes.
    * A few things from QPDFObjectHandle that used to be exceptions now write errors with the default logger.
    * By configuring the default logger, it is possible to capture output and errors that slipped through the cracks with setOutputStreams.
    * A C API is available in include/qpdf/qpdflogger-c.h.
    * See examples examples/qpdfjob-save-attachment.cc and examples/qpdfjob-c-save-attachment.cc.
  - In QPDFObjectHandle, new methods insertItemAndGetNew, appendItemAndGetNew, and replaceKeyAndGetNew return the newly added item. New methods eraseItemAndGetOld, replaceKeyAndGetOld, and removeKeyAndGetOld return the item that was just removed or, in the case of replaceKeyAndGetOld, a null object if the object was not previously there.
  - The QPDFObjectHandle::isDestroyed method can be used to detect when an indirect object QPDFObjectHandle belongs to a QPDF that has been destroyed. Any attempt to unparse this type of QPDFObjectHandle will throw a logic error.
  - The QPDFObjectHandle::getOwningQPDF method now returns a null pointer rather than an invalid pointer when the owning QPDF object has been destroyed. Indirect objects whose owning QPDF has been destroyed become invalid. Direct objects just lose their owning QPDF but continue to be valid.
  - The method QPDFObjectHandle::getQPDF is an alternative to QPDFObjectHandle::getOwningQPDF. It returns a QPDF& rather than a QPDF* and can be used when the object is known to have an owning QPDF. It throws an exception if the object does not have an owning QPDF. Only indirect objects are guaranteed to have an owning QPDF. Direct objects may have one if they were initially read from a PDF input source that is still valid, but it's also possible to have direct objects that don't have an owning QPDF.
  - Add method QPDFObjectHandle::isSameObjectAs for testing whether two QPDFObjectHandle objects point to the same underlying object, meaning changes to one will be reflected in the other. Note that this method does not compare the contents of the objects, so two distinct but structurally identical objects will not be considered the same object.
  - New factory method QPDF::create() returns a std::shared_ptr<QPDF>.
  - New Pipeline methods have been added to reduce the amount of casting that is needed:

    - `write`: overloaded version that takes `char const*` in addition to the one that takes `unsigned char const*`
    - `writeCstr`: writes a null-terminated C string
    - `writeString`: writes a std::string
    - `operator <<`: for null-terminated C strings, std::strings, and integer types
  - New `Pipeline` type `Pl_OStream` writes to a `std::ostream`.
  - New `Pipeline` type `Pl_String` appends to a `std::string`.
  - New `Pipeline` type `Pl_Function` can be used to call an arbitrary function on write. It supports `std::function` for C++ code and can also accept C-style functions that indicate success using a return value and take an extra parameter for passing user data.
  - Methods have been added to `QUtil` for converting PDF timestamps and `QPDFTime` objects to ISO-8601 timestamps.
  - Enhance JSON class to better support incrementally reading and writing large amounts of data without having to keep everything in memory.
  - Add new functions to the C API for `qpdfjob` that use a `qpdfjob_handle`. Like with the regular C API for qpdf, you have to call `qpdfjob_init` first, pass the handle to the functions, and call `qpdfjob_cleanup` at the end. This interface offers more flexibility than the old interface, which remains available.
  - Add `QPDFJob::registerProgressReporter` and `qpdfjob_register_progress_reporter` to allow a custom progress reporter to be used with `QPDFJob`. The `QPDFJob` object must be configured to report progress (via command-line argument or otherwise) for this to be used.
  - Add new overloads to `QPDFObjectHandle::StreamDataProvider::provideStreamData` that take `QPDFObjGen const&` instead of separate object ID and generation parameters. The old versions will continue to be supported and are not deprecated.
  - In `QPDFPageObjectHelper`, add a `copy_if_fallback` parameter to most of the page bounding box methods, and clarify in the comments about the difference between `copy_if_shared` and `copy_if_fallback`.
  - Add a move constructor to the `Buffer` class.
- Other changes
  - On GitHub, the release tags are now *vX.Y.Z* instead of *release-qpdf-X.Y.Z* to be more consistent with current practice.
  - In JSON v1 mode, the `"objects"` key now reflects the repaired pages tree if `"pages"` (or any other key that has the side effect of repairing the page tree) is specified. To see the original objects with any unrepaired page tree errors, specify `"objects"` and/or `"objectinfo"` by themselves. This is consistent with how JSON v2 behaves.
  - A new chapter on contributing to qpdf has been added to the documentation. See *Contributing to qpdf*.
  - The qpdf source code is now formatted automatically with `clang-format`. See *Code Formatting* for information.
  - Test coverage with `QTC` is enabled during development but compiled out of distributed qpdf binaries by default. This results in a significant performance improvement, especially on Windows. `QTC::TC` is still available in the library and is still usable by end user code even though calls to it made internally by the library are turned off. Internally, there is some additional caching to reduce the overhead of repeatedly reading environment variables at runtime.

  - The test files used by the `performance_check` script at the top of the repository are now available in the qpdf/performance-test-files github repository. In addition to running time, memory usage is also included in performance test results when available. The `performance_check` tool has only been tested on Linux.
  - Lots of code cleanup and refactoring work was contributed in multiple pull requests by M. Holger. This includes the work required to enable detection of `QPDFObjectHandle` objects that belong to destroyed `QPDF` objects.

10.6.3: March 8, 2022

- Announcement of upcoming change:
  - qpdf 11 will be built with cmake. The qpdf 11 documentation will include detailed migration instructions.
- Bug fixes:
  - Recognize strings explicitly encoded as UTF-8 as allowed by the PDF 2.0 spec.
  - Fix edge cases with appearance stream generation for form fields whose `/DA` field lacks proper font size specification or that specifies auto sizing. At this time, qpdf does not support auto sizing.
  - Minor, non-functional changes to build and documentation to accommodate a wider range of compilation environments in preparation for migration to cmake.

10.6.2: February 16, 2022

- Bug fixes:
  - Recognize strings encoded as UTF-16LE as Unicode. The PDF spec only allows UTF-16BE, but most readers accept UTF16-LE as well.
  - Fix a regression in command-line argument parsing to restore a previously undocumented behavior that some people were relying on.
  - Fix one more problem with mapping Unicode to PDF doc encoding

10.6.1: February 11, 2022

- Fix compilation errors on some platforms

10.6.0: February 9, 2022

- Preparation for replacement of `PointerHolder`

  The next major release of qpdf will replace `PointerHolder` with `std::shared_ptr` across all of qpdf's public API. No action is required at this time, but if you'd like to prepare, read the comments in `include/qpdf/PointerHolder.hh` and see *Smart Pointers* for details on what you can do now to create code that will continue to work with older versions of qpdf and be easier to switch over to qpdf 11 when it comes out.

- Preparation for a new JSON output version
  - The *--json* option takes an optional parameter indicating the version of the JSON output. At present, there is only one JSON version (1), but there are plans for an updated version in a coming release. Until the release of qpdf 11, the default value of `--json` is 1 for compatibility. Once qpdf 11 is out, the default version will be `latest`. If you are depending on the exact format of `--json` for code, you should start using `--json=1` in preparation.
- New QPDFJob API exposes CLI functionality

  Prior to qpdf 10.6, a lot of the functionality implemented by the qpdf CLI executable was built into the executable itself and not available from the library. qpdf 10.6 introduces a new object, `QPDFJob`, that exposes all of the command-line functionality. This includes a native `QPDFJob` API with fluent interfaces

that mirror the command-line syntax, a JSON syntax for specifying the equivalent of a command-line invocation, and the ability to run a qpdf "job" by passing a null-terminated array of qpdf command-line options. The command-line argument array and JSON methods of invoking `QPDFJob` are also exposed to the C API. For details, see *QPDFJob: a Job-Based Interface*.

- Other Library Enhancements
  - New `QPDFObjectHandle` literal syntax using C++'s user-defined literal syntax. You can use

```
auto oh = "<</Some (valid) /PDF (object)>>"_qpdf;
```

    to create a QPDFObjectHandle. It is a shorthand for `QPDFObjectHandle::parse`.

  - Preprocessor symbols `QPDF_MAJOR_VERSION`, `QPDF_MINOR_VERSION`, and `QPDF_PATCH_VERSION` are now available and can be used to make it easier to write code that supports multiple versions of qpdf. You don't have to include any new header files to get these, which makes it possible to write code like this:

```
#if !defined(QPDF_MAJOR_VERSION) || QPDF_MAJOR_VERSION < 11
    // do something using qpdf 10 or older API
#else
    // do something using qpdf 11 or newer API
#endif
```

    Since this was introduced only in qpdf version 10.6.0, testing for an undefined value of `QPDF_MAJOR_VERSION` is equivalent to detecting a version prior to 10.6.0.

    The symbol `QPDF_VERSION` is also defined as a string containing the same version number that is returned by `QPDF::QPDFVersion`. Note that `QPDF_VERSION` may differ from `QPDF::QPDFVersion()` if your header files and library are out of sync with each other.

  - The method `QPDF::QPDFVersion` and corresponding C API call `qpdf_get_qpdf_version` are now both guaranteed to return a reference (or pointer) to a static string, so you don't have to copy these if you are using them in your software. They have always returned static values. Now the fact that they return static values is part of the API contract and can be safely relied upon.

  - New accessor methods for `QPDFObjectHandle`. In addition to the traditional ones, such as `getIntValue`, `getName`, etc., there are a family of new accessors whose names are of the form `getValueAsX`. The difference in behavior is as follows:

    * The older accessor methods, which will continue to be supported, return the value of the object if it is the expected type. Otherwise, they return a fallback value and issue a warning.

    * The newer accessor methods return a boolean indicating whether or not the object is of the expected type. If it is, a reference to a variable of the correct type is initialized.

    In many cases, the new interfaces will enable more compact code and will also never generate type warnings. Thanks to M. Holger for contributing these accessors. Search for `getValueAs` in `include/qpdf/QPDFObjectHandle.hh` for a complete list.

    These are also exposed in the C API in functions whose names start with `qpdf_oh_get_value_as`.

  - New convenience methods in `QPDFObjectHandle`: `isDictionaryOfType`, `isStreamOfType`, and `isNameAndEquals` allow more compact querying of dictionaries. Also added to the C API: `qpdf_oh_is_dictionary_of_type` and `qpdf_oh_is_name_and_equals`. Thanks to M. Holger for the contribution.

  - New convenience method in `QPDFObjectHandle`: `getKeyIfDict` returns null when called on null and otherwise calls `getKey`. This makes it easier to access optional, lower-level dictionaries. It is exposed in the C API `qpdf_oh_get_key_if_dict`. Thanks to M. Holger for the contribution.

  - New functions added to QUtil: make_shared_cstr and make_unique_cstr copy std::string to std::shared_ptr<char> and std::unique_ptr<char[]>. These are alternatives to the existing QUtil::copy_string function which offer other ways to get a C string with safer memory management.
  - New function QUtil::file_can_be_opened tests to see whether a file can actually be opened by attempting to open it and close it again.
  - There is a new version of QUtil::call_main_from_wmain that takes a const argv array and calls a main that takes a const argv array.
  - QPDF::emptyPDF has been exposed to the C API as qpdf_empty_pdf. This makes it possible to create a PDF from scratch with the C API.
  - New C API functions qpdf_oh_get_binary_utf8_value and qpdf_oh_new_binary_unicode_string take length parameters, which makes it possible to handle UTF-8-encoded C strings with embedded NUL characters. Thanks to M. Holger for the contribution.
  - There is a new PDFVersion class for representing a PDF version number with the ability to compare and order PDF versions. Methods QPDF::getVersionAsPDFVersion and a new version of QPDFWriter::setMinimumPDFVersion use it. This makes it easier to create an output file whose PDF version is the maximum of the versions across all the input files that contributed to it.
  - The JSON object in the qpdf library has been enhanced to include a parser and the ability to get values out of the JSON object. Previously it was a write-only interface. Even so, qpdf's JSON object is not intended to be a general-purpose JSON implementation as discussed in include/qpdf/JSON.hh.
  - The JSON object's "schema" checking functionality now allows for optional keys. Note that this "schema" functionality doesn't conform to any type of standard. It's just there to help with error reporting with qpdf's own JSON support.

- Documentation Enhancements
  - Documentation for the command-line tool has been completely rewritten. This includes a top-to-bottom rewrite of *Running qpdf* in the manual. Command-line arguments are now indexed, and internal links can appear to them within the documentation.
  - The output of qpdf --help is generated from the manual and is divided into help topics that parallel the sections of the manual. When you run qpdf --help, instead of getting a Great Wall of Text, you are given basic usage information and a list of help topics. It is possible to request help for any individual topic or any specific command-line option, or you can get a dump of all available help text. The manual continues to contain a greater level of detail and more examples.

- Bug Fixes
  - Some characters were not correctly translated from PDF doc encoding to Unicode.
  - When splitting or combining pages, ensure that all output files have a PDF version greater than or equal to the maximum version of all the input files.

10.5.0: December 21, 2021

- Packaging changes
  - Pre-built documentation is no longer distributed with the source distribution. The AppImage and Windows binary distributions still contain embedded documentation, and a separate doc distribution file is available from the qpdf release site. Documentation is now available at https://qpdf.readthedocs.io for every major/minor version starting with version 10.5. Please see *Packaging Documentation* for details on how packagers should handle documentation.

  - The documentation sources have been switched from docbook to reStructuredText processed with Sphinx. This will break previous documentation links. A redirect is in place on the main website. A top-to-bottom review of the documentation is planned for an upcoming release.

- Library Enhancements

  - Since qpdf version 8, using object accessor methods on an instance of `QPDFObjectHandle` may create warnings if the object is not of the expected type. These warnings now have an error code of `qpdf_e_object` instead of `qpdf_e_damaged_pdf`. Also, comments have been added to `QPDFObjectHandle.hh` to explain in more detail what the behavior is. See *Object Accessor Methods* for a more in-depth discussion.

  - Add `Pl_Buffer::getMallocBuffer()` to initialize a buffer allocated with `malloc()` for better cross-language interoperability.

- C API Enhancements

  - Many thanks to M. Holger whose contributions have heavily influenced these C API enhancements. His several suggestions, pull requests, questions, and critical reading of documentation and comments have resulted in significant usability improvements to the C API.

  - Overhaul error handling for the object handle functions C API. Some rare error conditions that would previously have caused a crash are now trapped and reported, and the functions that generate them return fallback values. See comments in the `ERROR HANDLING` section of `include/qpdf/qpdf-c.h` for details. In particular, exceptions thrown by the underlying C++ code when calling object accessors are caught and converted into errors. The errors can be checked by calling `qpdf_has_error`. Use `qpdf_silence_errors` to prevent the error from being written to stderr.

  - Add `qpdf_get_last_string_length` to the C API to get the length of the last string that was returned. This is needed to handle strings that contain embedded null characters.

  - Add `qpdf_oh_is_initialized` and `qpdf_oh_new_uninitialized` to the C API to make it possible to work with uninitialized objects.

  - Add `qpdf_oh_new_object` to the C API. This allows you to clone an object handle.

  - Add `qpdf_get_object_by_id`, `qpdf_make_indirect_object`, and `qpdf_replace_object`, exposing the corresponding methods in `QPDF` and `QPDFObjectHandle`.

  - Add several functions for working with pages. See `PAGE FUNCTIONS` in `include/qpdf/qpdf-c.h` for details.

  - Add several functions for working with streams. See `STREAM FUNCTIONS` in `include/qpdf/qpdf-c.h` for details.

  - Add `qpdf_oh_get_type_code` and `qpdf_oh_get_type_name`.

  - Add `qpdf_oh_get_binary_string_value` and `qpdf_oh_new_binary_string` for making it easier to deal with strings that contain embedded null characters.

10.4.0: November 16, 2021

- Handling of Weak Cryptography Algorithms

  - From the qpdf CLI, the `--allow-weak-crypto` is now required to suppress a warning when explicitly creating PDF files using RC4 encryption. While qpdf will always retain the ability to read and write such files, doing so will require explicit acknowledgment moving forward. For qpdf 10.4, this change only affects the command-line tool. Starting in qpdf 11, there will be small API changes to require explicit acknowledgment in those cases as well. For additional information, see *Weak Cryptography*.

- Bug Fixes

  - Fix potential bounds error when handling shell completion that could occur when given bogus input.

  - Properly handle overlay/underlay on completely empty pages (with no resource dictionary).
  - Fix crash that could occur under certain conditions when using *--pages* with files that had form fields.
- Library Enhancements
  - Make `QPDF::findPage` functions public.
  - Add methods to `Pl_Flate` to be able to receive warnings on certain recoverable conditions.
  - Add an extra check to the library to detect when foreign objects are inserted directly (instead of using `QPDF::copyForeignObject`) at the time of insertion rather than when the file is written. Catching the error sooner makes it much easier to locate the incorrect code.
- CLI Enhancements
  - Improve diagnostics around parsing *--pages* command-line options
- Packaging Changes
  - The Windows binary distribution is now built with crypto provided by OpenSSL 3.0.

10.3.2: May 8, 2021

- Bug Fixes
  - When generating a file while preserving object streams, unreferenced objects are correctly removed unless *--preserve-unreferenced* is specified.
- Library Enhancements
  - When adding a page that already exists, make a shallow copy instead of throwing an exception. This makes the library behavior consistent with the CLI behavior. See `ChangeLog` for additional notes.

10.3.1: March 11, 2021

- Bug Fixes
  - Form field copying failed on files where /DR was a direct object in the document-level form dictionary.

10.3.0: March 4, 2021

- Bug Fixes
  - The code for handling form fields when copying pages from 10.2.0 was not quite right and didn't work in a number of situations, such as when the same page was copied multiple times or when there were conflicting resource or field names across multiple copies. The 10.3.0 code has been much more thoroughly tested with more complex cases and with a multitude of readers and should be much closer to correct. The 10.2.0 code worked well enough for page splitting or for copying pages with form fields into documents that didn't already have them but was still not quite correct in handling of field-level resources.
  - When `QPDF::replaceObject` or `QPDF::swapObjects` is called, existing `QPDFObjectHandle` instances no longer point to the old objects. The next time they are accessed, they automatically notice the change to the underlying object and update themselves. This resolves a very longstanding source of confusion, albeit in a very rarely used method call.
  - Fix form field handling code to look for default appearances, quadding, and default resources in the right places. The code was not looking for things in the document-level interactive form dictionary that it was supposed to be finding there. This required adding a few new methods to `QPDFFormFieldObjectHelper`.
- Library Enhancements

- Reworked the code that handles copying annotations and form fields during page operations. There were additional methods added to the public API from 10.2.0 and a one deprecation of a method added in 10.2.0. The majority of the API changes are in methods most people would never call and that will hopefully be superseded by higher-level interfaces for handling page copies. Please see the `ChangeLog` file for details.
- The method `QPDF::numWarnings` was added so that you can tell whether any warnings happened during a specific block of code.

10.2.0: February 23, 2021

- CLI Behavior Changes
  - Operations that work on combining pages are much better about protecting form fields. In particular, *--split-pages* and *--pages* now preserve interaction form functionality by copying the relevant form field information from the original files. Additionally, if you use *--pages* to select only some pages from the original input file, unused form fields are removed, which prevents lots of unused annotations from being retained.
  - By default, **qpdf** no longer allows creation of encrypted PDF files whose user password is non-empty and owner password is empty when a 256-bit key is in use. The *--allow-insecure* option, specified inside the *--encrypt* options, allows creation of such files. Behavior changes in the CLI are avoided when possible, but an exception was made here because this is security-related. qpdf must always allow creation of weird files for testing purposes, but it should not default to letting users unknowingly create insecure files.
- Library Behavior Changes
  - Note: the changes in this section cause differences in output in some cases. These differences change the syntax of the PDF but do not change the semantics (meaning). I make a strong effort to avoid gratuitous changes in qpdf's output so that qpdf changes don't break people's tests. In this case, the changes significantly improve the readability of the generated PDF and don't affect any output that's generated by simple transformation. If you are annoyed by having to update test files, please rest assured that changes like this have been and will continue to be rare events.
  - `QPDFObjectHandle::newUnicodeString` now uses whichever of ASCII, PDFDocEncoding, of UTF-16 is sufficient to encode all the characters in the string. This reduces needless encoding in UTF-16 of strings that can be encoded in ASCII. This change may cause qpdf to generate different output than before when form field values are set using `QPDFFormFieldObjectHelper` but does not change the meaning of the output.
  - The code that places form XObjects and also the code that flattens rotations trim trailing zeroes from real numbers that they calculate. This causes slight (but semantically equivalent) differences in generated appearance streams and form XObject invocations in overlay/underlay code or in user code that calls the methods that place form XObjects on a page.
- CLI Enhancements
  - Add new command line options for listing, saving, adding, removing, and and copying file attachments. See *Embedded Files/Attachments* for details.
  - Page splitting and merging operations, as well as *--flatten-rotation*, are better behaved with respect to annotations and interactive form fields. In most cases, interactive form field functionality and proper formatting and functionality of annotations is preserved by these operations. There are still some cases that aren't perfect, such as when functionality of annotations depends on document-level data that qpdf doesn't yet understand or when there are problems with referential integrity among form fields and annotations (e.g., when a single form field object or its associated annotations are shared across multiple pages, a case that is out of spec but that works in most viewers anyway).

  - The option `--password-file=filename` can now be used to read the decryption password from a file. You can use `-` as the file name to read the password from standard input. This is an easier/more obvious way to read passwords from files or standard input than using `@file` for this purpose.
  - Add some information about attachments to the JSON output, and added `attachments` as an additional JSON key. The information included here is limited to the preferred name and content stream and a reference to the file spec object. This is enough detail for clients to avoid the hassle of navigating a name tree and provides what is needed for basic enumeration and extraction of attachments. More detailed information can be obtained by following the reference to the file spec object.
  - Add numeric option to `--collate`. If `--collate=n` is given, take pages in groups of `n` from the given files.
  - It is now valid to provide `--rotate=0` to clear rotation from a page.

- Library Enhancements
  - This release includes numerous additions to the API. Not all changes are listed here. Please see the `ChangeLog` file in the source distribution for a comprehensive list. Highlights appear below.
  - Add `QPDFObjectHandle::ditems()` and `QPDFObjectHandle::aitems()` that enable C++-style iteration, including range-for iteration, over dictionary and array QPDFObjectHandles. See comments in `include/qpdf/QPDFObjectHandle.hh` and `examples/pdf-name-number-tree.cc` for details.
  - Add `QPDFObjectHandle::copyStream` for making a copy of a stream within the same `QPDF` instance.
  - Add new helper classes for supporting file attachments, also known as embedded files. New classes are `QPDFEmbeddedFileDocumentHelper`, `QPDFFileSpecObjectHelper`, and `QPDFEFStreamObjectHelper`. See their respective headers for details and `examples/pdf-attach-file.cc` for an example.
  - Add a version of `QPDFObjectHandle::parse` that takes a `QPDF` pointer as context so that it can parse strings containing indirect object references. This is illustrated in `examples/pdf-attach-file.cc`.
  - Re-implement `QPDFNameTreeObjectHelper` and `QPDFNumberTreeObjectHelper` to be more efficient, add an iterator-based API, give them the capability to repair broken trees, and create methods for modifying the trees. With this change, qpdf has a robust read/write implementation of name and number trees.
  - Add new versions of `QPDFObjectHandle::replaceStreamData` that take `std::function` objects for cases when you need something between a static string and a full-fledged StreamDataProvider. Using this with `QUtil::file_provider` is a very easy way to create a stream from the contents of a file.
  - The `QPDFMatrix` class, formerly a private, internal class, has been added to the public API. See `include/qpdf/QPDFMatrix.hh` for details. This class is for working with transformation matrices. Some methods in `QPDFPageObjectHelper` make use of this to make information about transformation matrices available. For an example, see `examples/pdf-overlay-page.cc`.
  - Several new methods were added to `QPDFAcroFormDocumentHelper` for adding, removing, getting information about, and enumerating form fields.
  - Add method `QPDFAcroFormDocumentHelper::transformAnnotations`, which applies a transformation to each annotation on a page.
  - Add `QPDFPageObjectHelper::copyAnnotations`, which copies annotations and, if applicable, associated form fields, from one page to another, possibly transforming the rectangles.

- Build Changes

  - A C++-14 compiler is now required to build qpdf. There is no intention to require anything newer than that for a while. C++-14 includes modest enhancements to C++-11 and appears to be supported about as widely as C++-11.

- Bug Fixes
  - The *--flatten-rotation* option applies transformations to any annotations that may be on the page.
  - If a form XObject lacks a resources dictionary, consider any names in that form XObject to be referenced from the containing page. This is compliant with older PDF versions. Also detect if any form XObjects have any unresolved names and, if so, don't remove unreferenced resources from them or from the page that contains them. Unfortunately this has the side effect of preventing removal of unreferenced resources in some cases where names appear that don't refer to resources, such as with tagged PDF. This is a bit of a corner case that is not likely to cause a significant problem in practice, but the only side effect would be lack of removal of shared resources. A future version of qpdf may be more sophisticated in its detection of names that refer to resources.
  - Properly handle strings if they appear in inline image dictionaries while externalizing inline images.

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.
  - 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 rect-

angle. 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 *Crypto Providers* for more information about crypto providers and *Build-time Crypto Selection* 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 *Crypto Providers*.
- CLI Enhancements
  - Addition of the *`--show-crypto`* option in support of selectable crypto providers, as described in *Crypto Providers*.
  - 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* 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 *Casting Policy* 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, getIntValueAsInt, 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. 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 *Unicode Passwords*. 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 *Unicode Passwords*.

  - 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.
  - Add options `--overlay` and `--underlay` for overlaying or underlaying pages of other files onto output pages. See *Overlay and Underlay* 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 *Unicode Passwords* 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 *Unicode Passwords* 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.
- Exit codes returned by `QPDFJob::run()` and the C API wrappers are now defined in `qpdf/Constants.h` in the `qpdf_exit_code_e` type so that they are accessible from the C API. They were previously only defined as constants in `qpdf/QPDFJob.hh`.

- 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 *Page Selection* 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 *qpdf JSON*.
  - 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 `QPDFObjectHandle::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, 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 *Notes for Packagers*.
  - 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`* 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*.

- 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 *Helper Classes*. 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.

8.0.1: March 4, 2018

- Disregard data check errors when uncompressing `/FlateDecode` streams. This is consistent with most other PDF readers and allows qpdf to recover data from another class of malformed PDF files.
- On the command line when specifying page ranges, support preceding a page number by "r" to indicate that it should be counted from the end. For example, the range `r3-r1` would indicate the last three pages of a document.

8.0.0: February 25, 2018

- Packaging and Distribution Changes
  - QPDF is now distributed as an AppImage in addition to all the other ways it is distributed. The AppImage can be found in the download area with the other packages. Thanks to Kurt Pfeifle and Simon Peter for their contributions.
- Bug Fixes
  - `QPDFObjectHandle::getUTF8Val` now properly treats non-Unicode strings as encoded with PDF Doc Encoding.
  - Improvements to handling of objects in PDF files that are not of the expected type. In most cases, qpdf will be able to warn for such cases rather than fail with an exception. Previous versions of qpdf would sometimes fail with errors such as "operation for dictionary object attempted on object of wrong type". This situation should be mostly or entirely eliminated now.
- Enhancements to the **qpdf** Command-line Tool. All new options listed here are documented in more detail in *Running qpdf*.
  - The option `--linearize-pass1=`*`file`* has been added for debugging qpdf's linearization code.
  - The option *`--coalesce-contents`* can be used to combine content streams of a page whose contents are an array of streams into a single stream.
- API Enhancements. All new API calls are documented in their respective classes' header files. There are no non-compatible changes to the API.
  - Add function `qpdf_check_pdf` to the C API. This function does basic checking that is a subset of what **qpdf --check** performs.
  - Major enhancements to the lexical layer of qpdf. For a complete list of enhancements, please refer to the `ChangeLog` file. Most of the changes result in improvements to qpdf's ability handle erroneous files. It is also possible for programs to handle whitespace, comments, and inline images as tokens.
  - New API for working with PDF content streams at a lexical level. The new class `QPDFObjectHandle::TokenFilter` allows the developer to provide token handlers. Token filters can be used with several different methods in `QPDFObjectHandle` as well as with a lower-level interface. See comments in `QPDFObjectHandle.hh` as well as the new examples `examples/pdf-filter-tokens.cc` and `examples/pdf-count-strings.cc` for details.

7.1.1: February 4, 2018

- Bug fix: files whose /ID fields were other than 16 bytes long can now be properly linearized
- A few compile and link issues have been corrected for some platforms.

7.1.0: January 14, 2018

- PDF files contain streams that may be compressed with various compression algorithms which, in some cases, may be enhanced by various predictor functions. Previously only the PNG up predictor was supported. In this version, all the PNG predictors as well as the TIFF predictor are supported. This increases the range of files that qpdf is able to handle.
- QPDF now allows a raw encryption key to be specified in place of a password when opening encrypted files, and will optionally display the encryption key used by a file. This is a non-standard operation, but it can be useful in certain situations. Please see the discussion of `--password-is-hex-key` or the comments around `QPDF::setPasswordIsHexKey` in `QPDF.hh` for additional details.
- Bug fix: numbers ending with a trailing decimal point are now properly recognized as numbers.
- Bug fix: when building qpdf from source on some platforms (especially MacOS), the build could get confused by older versions of qpdf installed on the system. This has been corrected.

7.0.0: September 15, 2017

- Packaging and Distribution Changes
  - QPDF's primary license is now version 2.0 of the Apache License rather than version 2.0 of the Artistic License. You may still, at your option, consider qpdf to be licensed with version 2.0 of the Artistic license.
  - QPDF no longer has a dependency on the PCRE (Perl-Compatible Regular Expression) library. QPDF now has an added dependency on the JPEG library.
- Bug Fixes
  - This release contains many bug fixes for various infinite loops, memory leaks, and other memory errors that could be encountered with specially crafted or otherwise erroneous PDF files.
- New Features
  - QPDF now supports reading and writing streams encoded with JPEG or RunLength encoding. Library API enhancements and command-line options have been added to control this behavior. See command-line options `--compress-streams` and `--decode-level` and methods `QPDFWriter::setCompressStreams` and `QPDFWriter::setDecodeLevel`.
  - QPDF is much better at recovering from broken files. In most cases, qpdf will skip invalid objects and will preserve broken stream data by not attempting to filter broken streams. QPDF is now able to recover or at least not crash on dozens of broken test files I have received over the past few years.
  - Page rotation is now supported and accessible from both the library and the command line.
  - `QPDFWriter` supports writing files in a way that preserves PCLm compliance in support of driverless printing. This is very specialized and is only useful to applications that already know how to create PCLm files.
- Enhancements to the **qpdf** Command-line Tool. All new options listed here are documented in more detail in *Running qpdf*.
  - Command-line arguments can now be read from files or standard input using `@file` or `@-` syntax. Please see *Basic Invocation*.
  - `--rotate`: request page rotation

  - *--newline-before-endstream*: ensure that a newline appears before every `endstream` keyword in the file; used to prevent qpdf from breaking PDF/A compliance on already compliant files.
  - *--preserve-unreferenced*: preserve unreferenced objects in the input PDF
  - *--split-pages*: break output into chunks with fixed numbers of pages
  - *--verbose*: print the name of each output file that is created
  - *--compress-streams* and *--decode-level* replace *--stream-data* for improving granularity of controlling compression and decompression of stream data. The *--stream-data* option will remain available.
  - When running **qpdf --check** with other options, checks are always run first. This enables qpdf to perform its full recovery logic before outputting other information. This can be especially useful when manually recovering broken files, looking at qpdf's regenerated cross reference table, or other similar operations.
  - Process **--pages** earlier so that other options like *--show-pages* or *--split-pages* can operate on the file after page splitting/merging has occurred.

- API Changes. All new API calls are documented in their respective classes' header files.
  - `QPDFObjectHandle::rotatePage`: apply rotation to a page object
  - `QPDFWriter::setNewlineBeforeEndstream`: force newline to appear before `endstream`
  - `QPDFWriter::setPreserveUnreferencedObjects`: preserve unreferenced objects that appear in the input PDF. The default behavior is to discard them.
  - New `Pipeline` types `Pl_RunLength` and `Pl_DCT` are available for developers who wish to produce or consume RunLength or DCT stream data directly. The `examples/pdf-create.cc` example illustrates their use.
  - `QPDFWriter::setCompressStreams` and `QPDFWriter::setDecodeLevel` methods control handling of different types of stream compression.
  - Add new C API functions `qpdf_set_compress_streams`, `qpdf_set_decode_level`, `qpdf_set_preserve_unreferenced_objects`, and `qpdf_set_newline_before_endstream` corresponding to the new `QPDFWriter` methods.

6.0.0: November 10, 2015

- Implement *--deterministic-id* command-line option and `QPDFWriter::setDeterministicID` as well as C API function `qpdf_set_deterministic_ID` for generating a deterministic ID for non-encrypted files. When this option is selected, the ID of the file depends on the contents of the output file, and not on transient items such as the timestamp or output file name.
- Make qpdf more tolerant of files whose xref table entries are not the correct length.

5.1.3: May 24, 2015

- Bug fix: fix-qdf was not properly handling files that contained object streams with more than 255 objects in them.
- Bug fix: qpdf was not properly initializing Microsoft's secure crypto provider on fresh Windows installations that had not had any keys created yet.
- Fix a few errors found by Gynvael Coldwind and Mateusz Jurczyk of the Google Security Team. Please see the ChangeLog for details.
- Properly handle pages that have no contents at all. There were many cases in which qpdf handled this fine, but a few methods blindly obtained page contents with handling the possibility that there were no contents.
- Make qpdf more robust for a few more kinds of problems that may occur in invalid PDF files.

5.1.2: June 7, 2014

- Bug fix: linearizing files could create a corrupted output file under extremely unlikely file size circumstances. See ChangeLog for details. The odds of getting hit by this are very low, though one person did.
- Bug fix: qpdf would fail to write files that had streams with decode parameters referencing other streams.
- New example program: **pdf-split-pages**: efficiently split PDF files into individual pages. The example program does this more efficiently than using **qpdf --pages** to do it.
- Packaging fix: Visual C++ binaries did not support Windows XP. This has been rectified by updating the compilers used to generate the release binaries.

5.1.1: January 14, 2014

- Performance fix: copying foreign objects could be very slow with certain types of files. This was most likely to be visible during page splitting and was due to traversing the same objects multiple times in some cases.

5.1.0: December 17, 2013

- Added runtime option (`QUtil::setRandomDataProvider`) to supply your own random data provider. You can use this if you want to avoid using the OS-provided secure random number generation facility or stdlib's less secure version. See comments in include/qpdf/QUtil.hh for details.
- Fixed image comparison tests to not create 12-bit-per-pixel images since some versions of tiffcmp have bugs in comparing them in some cases. This increases the disk space required by the image comparison tests, which are off by default anyway.
- Introduce a number of small fixes for compilation on the latest clang in MacOS and the latest Visual C++ in Windows.
- Be able to handle broken files that end the xref table header with a space instead of a newline.

5.0.1: October 18, 2013

- Thanks to a detailed review by Florian Weimer and the Red Hat Product Security Team, this release includes a number of non-user-visible security hardening changes. Please see the ChangeLog file in the source distribution for the complete list.
- When available, operating system-specific secure random number generation is used for generating initialization vectors and other random values used during encryption or file creation. For the Windows build, this results in an added dependency on Microsoft's cryptography API. To disable the OS-specific cryptography and use the old version, pass the `--enable-insecure-random` option to **./configure**.
- The **qpdf** command-line tool now issues a warning when `-accessibility=n` is specified for newer encryption versions stating that the option is ignored. qpdf, per the spec, has always ignored this flag, but it previously did so silently. This warning is issued only by the command-line tool, not by the library. The library's handling of this flag is unchanged.

5.0.0: July 10, 2013

- Bug fix: previous versions of qpdf would lose objects with generation != 0 when generating object streams. Fixing this required changes to the public API.
- Removed methods from public API that were only supposed to be called by QPDFWriter and couldn't realistically be called anywhere else. See ChangeLog for details.
- New `QPDFObjGen` class added to represent an object ID/generation pair. `QPDFObjectHandle::getObjGen()` is now preferred over `QPDFObjectHandle::getObjectID()` and `QPDFObjectHandle::getGeneration()` as it makes it less likely for people to accidentally write code that ignores the generation number. See `QPDF.hh` and `QPDFObjectHandle.hh` for additional notes.
- Add *--show-npages* command-line option to the **qpdf** command to show the number of pages in a file.

- Allow omission of the page range within *--pages* for the **qpdf** command. When omitted, the page range is implicitly taken to be all the pages in the file.
- Various enhancements were made to support different types of broken files or broken readers. Details can be found in ChangeLog.

4.1.0: April 14, 2013

- Note to people including qpdf in distributions: the .la files generated by libtool are now installed by qpdf's **make install** target. Before, they were not installed. This means that if your distribution does not want to include .la files, you must remove them as part of your packaging process.
- Major enhancement: API enhancements have been made to support parsing of content streams. This enhancement includes the following changes:
  - QPDFObjectHandle::parseContentStream method parses objects in a content stream and calls handlers in a callback class. The example examples/pdf-parse-content.cc illustrates how this may be used.
  - QPDFObjectHandle can now represent operators and inline images, object types that may only appear in content streams.
  - Method QPDFObjectHandle::getTypeCode() returns an enumerated type value representing the underlying object type. Method QPDFObjectHandle::getTypeName() returns a text string describing the name of the type of a QPDFObjectHandle object. These methods can be used for more efficient parsing and debugging/diagnostic messages.
- **qpdf --check** now parses all pages' content streams in addition to doing other checks. While there are still many types of errors that cannot be detected, syntactic errors in content streams will now be reported.
- Minor compilation enhancements have been made to facilitate easier for support for a broader range of compilers and compiler versions.
  - Warning flags have been moved into a separate variable in autoconf.mk
  - The configure flag --enable-werror work for Microsoft compilers
  - All MSVC CRT security warnings have been resolved.
  - All C-style casts in C++ Code have been replaced by C++ casts, and many casts that had been included to suppress higher warning levels for some compilers have been removed, primarily for clarity. Places where integer type coercion occurs have been scrutinized. A new casting policy has been documented in the manual. This is of concern mainly to people porting qpdf to new platforms or compilers. It is not visible to programmers writing code that uses the library
  - Some internal limits have been removed in code that converts numbers to strings. This is largely invisible to users, but it does trigger a bug in some older versions of mingw-w64's C++ library. See README-windows.md in the source distribution if you think this may affect you. The copy of the DLL distributed with qpdf's binary distribution is not affected by this problem.
- The RPM spec file previously included with qpdf has been removed. This is because virtually all Linux distributions include qpdf now that it is a dependency of CUPS filters.
- A few bug fixes are included:
  - Overridden compressed objects are properly handled. Before, there were certain constructs that could cause qpdf to see old versions of some objects. The most usual manifestation of this was loss of filled in form values for certain files.
  - Installation no longer uses GNU/Linux-specific versions of some commands, so **make install** works on Solaris with native tools.
  - The 64-bit mingw Windows binary package no longer includes a 32-bit DLL.

4.0.1: January 17, 2013

- Fix detection of binary attachments in test suite to avoid false test failures on some platforms.
- Add clarifying comment in `QPDF.hh` to methods that return the user password explaining that it is no longer possible with newer encryption formats to recover the user password knowing the owner password. In earlier encryption formats, the user password was encrypted in the file using the owner password. In newer encryption formats, a separate encryption key is used on the file, and that key is independently encrypted using both the user password and the owner password.

4.0.0: December 31, 2012

- Major enhancement: support has been added for newer encryption schemes supported by version X of Adobe Acrobat. This includes use of 127-character passwords, 256-bit encryption keys, and the encryption scheme specified in ISO 32000-2, the PDF 2.0 specification. This scheme can be chosen from the command line by specifying use of 256-bit keys. qpdf also supports the deprecated encryption method used by Acrobat IX. This encryption style has known security weaknesses and should not be used in practice. However, such files exist "in the wild," so support for this scheme is still useful. New methods `QPDFWriter::setR6EncryptionParameters` (for the PDF 2.0 scheme) and `QPDFWriter::setR5EncryptionParameters` (for the deprecated scheme) have been added to enable these new encryption schemes. Corresponding functions have been added to the C API as well.
- Full support for Adobe extension levels in PDF version information. Starting with PDF version 1.7, corresponding to ISO 32000, Adobe adds new functionality by increasing the extension level rather than increasing the version. This support includes addition of the `QPDF::getExtensionLevel` method for retrieving the document's extension level, addition of versions of `QPDFWriter::setMinimumPDFVersion` and `QPDFWriter::forcePDFVersion` that accept an extension level, and extended syntax for specifying forced and minimum versions on the command line as described in *--force-version* and *--min-version*. Corresponding functions have been added to the C API as well.
- Minor fixes to prevent qpdf from referencing objects in the file that are not referenced in the file's overall structure. Most files don't have any such objects, but some files have contain unreferenced objects with errors, so these fixes prevent qpdf from needlessly rejecting or complaining about such objects.
- Add new generalized methods for reading and writing files from/to programmer-defined sources. The method `QPDF::processInputSource` allows the programmer to use any input source for the input file, and `QPDFWriter::setOutputPipeline` allows the programmer to write the output file through any pipeline. These methods would make it possible to perform any number of specialized operations, such as accessing external storage systems, creating bindings for qpdf in other programming languages that have their own I/O systems, etc.
- Add new method `QPDF::getEncryptionKey` for retrieving the underlying encryption key used in the file.
- This release includes a small handful of non-compatible API changes. While effort is made to avoid such changes, all the non-compatible API changes in this version were to parts of the API that would likely never be used outside the library itself. In all cases, the altered methods or structures were parts of the `QPDF` that were public to enable them to be called from either `QPDFWriter` or were part of validation code that was over-zealous in reporting problems in parts of the file that would not ordinarily be referenced. In no case did any of the removed methods do anything worse that falsely report error conditions in files that were broken in ways that didn't matter. The following public parts of the `QPDF` class were changed in a non-compatible way:
  - Updated nested `QPDF::EncryptionData` class to add fields needed by the newer encryption formats, member variables changed to private so that future changes will not require breaking backward compatibility.
  - Added additional parameters to `compute_data_key`, which is used by `QPDFWriter` to compute the encryption key used to encrypt a specific object.

  - Removed the method `flattenScalarReferences`. This method was previously used prior to writing a new PDF file, but it has the undesired side effect of causing qpdf to read objects in the file that were not referenced. Some otherwise files have unreferenced objects with errors in them, so this could cause qpdf to reject files that would be accepted by virtually all other PDF readers. In fact, qpdf relied on only a very small part of what flattenScalarReferences did, so only this part has been preserved, and it is now done directly inside `QPDFWriter`.
  - Removed the method `decodeStreams`. This method was used by the *--check* option of the **qpdf** command-line tool to force all streams in the file to be decoded, but it also suffered from the problem of opening otherwise unreferenced streams and thus could report false positive. The *--check* option now causes qpdf to go through all the motions of writing a new file based on the original one, so it will always reference and check exactly those parts of a file that any ordinary viewer would check.
  - Removed the method `trimTrailerForWrite`. This method was used by `QPDFWriter` to modify the original QPDF object by removing fields from the trailer dictionary that wouldn't apply to the newly written file. This functionality, though generally harmless, was a poor implementation and has been replaced by having QPDFWriter filter these out when copying the trailer rather than modifying the original QPDF object. (Note that qpdf never modifies the original file itself.)

- Allow the PDF header to appear anywhere in the first 1024 bytes of the file. This is consistent with what other readers do.
- Fix the **pkg-config** files to list zlib and pcre in `Requires.private` to better support static linking using **pkg-config**.

3.0.2: September 6, 2012

- Bug fix: `QPDFWriter::setOutputMemory` did not work when not used with `QPDFWriter::setStaticID`, which made it pretty much useless. This has been fixed.
- New API call `QPDFWriter::setExtraHeaderText` inserts additional text near the header of the PDF file. The intended use case is to insert comments that may be consumed by a downstream application, though other use cases may exist.

3.0.1: August 11, 2012

- Version 3.0.0 included addition of files for **pkg-config**, but this was not mentioned in the release notes. The release notes for 3.0.0 were updated to mention this.
- Bug fix: if an object stream ended with a scalar object not followed by space, qpdf would incorrectly report that it encountered a premature EOF. This bug has been in qpdf since version 2.0.

3.0.0: August 2, 2012

- Acknowledgment: I would like to express gratitude for the contributions of Tobias Hoffmann toward the release of qpdf version 3.0. He is responsible for most of the implementation and design of the new API for manipulating pages, and contributed code and ideas for many of the improvements made in version 3.0. Without his work, this release would certainly not have happened as soon as it did, if at all.
- *Non-compatible API changes:*
  - The method `QPDFObjectHandle::replaceStreamData` that uses a `StreamDataProvider` to provide the stream data no longer takes a `length` parameter. The parameter was removed since this provides the user an opportunity to simplify the calling code. This method was introduced in version 2.2. At the time, the `length` parameter was required in order to ensure that calls to the stream data provider returned the same length for a specific stream every time they were invoked. In particular, the linearization code depends on this. Instead, qpdf 3.0 and newer check for that constraint explicitly. The first time the stream data provider is called for a specific stream, the actual length is saved, and subsequent calls are required to return the same number of bytes. This means the calling code no longer has to compute the length in advance, which can be a significant simplification. If your code

fails to compile because of the extra argument and you don't want to make other changes to your code, just omit the argument.

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

- Support has been added for large files. The test suite verifies support for files larger than 4 gigabytes, and manual testing has verified support for files larger than 10 gigabytes. Large file support is available for both 32-bit and 64-bit platforms as long as the compiler and underlying platforms support it.
- Support for page selection (splitting and merging PDF files) has been added to the **qpdf** command-line tool. See *Page Selection*.
- The *--copy-encryption* option have been added to the **qpdf** command-line tool for copying encryption parameters from another file.
- New methods have been added to the `QPDF` object for adding and removing pages. See *Adding and Removing Pages*.
- New methods have been added to the `QPDF` object for copying objects from other PDF files. See *Copying Objects From Other PDF Files*
- A new method `QPDFObjectHandle::parse` has been added for constructing `QPDFObjectHandle` objects from a string description.
- Methods have been added to `QPDFWriter` to allow writing to an already open stdio `FILE*` addition to writing to standard output or a named file. Methods have been added to `QPDF` to be able to process a file from an already open stdio `FILE*`. This makes it possible to read and write PDF from secure temporary files that have been unlinked prior to being fully read or written.
- The `QPDF::emptyPDF` can be used to allow creation of PDF files from scratch. The example `examples/pdf-create.cc` illustrates how it can be used.
- Several methods to take `PointerHolder<Buffer>` can now also accept `std::string` arguments.
- Many new convenience methods have been added to the library, most in `QPDFObjectHandle`. See `ChangeLog` for a full list.
- When building on a platform that supports ELF shared libraries (such as Linux), symbol versions are enabled by default. They can be disabled by passing `--disable-ld-version-script` to **./configure**.
- The file `libqpdf.pc` is now installed to support **pkg-config**.
- Image comparison tests are off by default now since they are not needed to verify a correct build or port of qpdf. They are needed only when changing the actual PDF output generated by qpdf. You should enable them if you are making deep changes to qpdf itself. See `README.md` for details.
- Large file tests are off by default but can be turned on with **./configure** or by setting an environment variable before running the test suite. See `README.md` for details.
- When qpdf's test suite fails, failures are not printed to the terminal anymore by default. Instead, find them in `build/qtest.log`. For packagers who are building with an autobuilder, you can add the `--enable-show-failed-test-output` option to **./configure** to restore the old behavior.

2.3.1: December 28, 2011

- Fix thread-safety problem resulting from non-thread-safe use of the PCRE library.
- Made a few minor documentation fixes.
- Add workaround for a bug that appears in some versions of ghostscript to the test suite
- Fix minor build issue for Visual C++ 2010.

2.3.0: August 11, 2011

- Bug fix: when preserving existing encryption on encrypted files with cleartext metadata, older qpdf versions would generate password-protected files with no valid password. This operation now works. This bug only affected files created by copying existing encryption parameters; explicit encryption with specification of cleartext metadata worked before and continues to work.
- Enhance `QPDFWriter` with a new constructor that allows you to delay the specification of the output file. When using this constructor, you may now call `QPDFWriter::setOutputFilename` to specify the output file, or you may use `QPDFWriter::setOutputMemory` to cause `QPDFWriter` to write the resulting PDF file to a memory buffer. You may then use `QPDFWriter::getBuffer` to retrieve the memory buffer.
- Add new API call `QPDF::replaceObject` for replacing objects by object ID
- Add new API call `QPDF::swapObjects` for swapping two objects by object ID
- Add `QPDFObjectHandle::getDictAsMap` and `QPDFObjectHandle::getArrayAsVector` to allow retrieval of dictionary objects as maps and array objects as vectors.
- Add functions `qpdf_get_info_key` and `qpdf_set_info_key` to the C API for manipulating string fields of the document's `/Info` dictionary.
- Add functions `qpdf_init_write_memory`, `qpdf_get_buffer_length`, and `qpdf_get_buffer` to the C API for writing PDF files to a memory buffer instead of a file.

2.2.4: June 25, 2011

- Fix installation and compilation issues; no functionality changes.

2.2.3: April 30, 2011

- Handle some damaged streams with incorrect characters following the stream keyword.
- Improve handling of inline images when normalizing content streams.
- Enhance error recovery to properly handle files that use object 0 as a regular object, which is specifically disallowed by the spec.

2.2.2: October 4, 2010

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

2.2.1: October 1, 2010

- Add new method `QPDF::setOutputStreams` to replace `std::cout` and `std::cerr` with other streams for generation of diagnostic messages and error messages. This can be useful for GUIs or other applications that want to capture any output generated by the library to present to the user in some other way. Note that QPDF does not write to `std::cout` (or the specified output stream) except where explicitly mentioned in `QPDF.hh`, and that the only use of the error stream is for warnings. Note also that output of warnings is suppressed when `setSuppressWarnings(true)` is called.
- Add new method `QPDF::processMemoryFile` for operating on PDF files that are loaded into memory rather than in a file on disk.
- Give a warning but otherwise ignore empty PDF objects by treating them as null. Empty object are not permitted by the PDF specification but have been known to appear in some actual PDF files.
- Handle inline image filter abbreviations when the appear as stream filter abbreviations. The PDF specification does not allow use of stream filter abbreviations in this way, but Adobe Reader and some other PDF readers accept them since they sometimes appear incorrectly in actual PDF files.
- Implement miscellaneous enhancements to `PointerHolder` and `Buffer` to support other changes.

2.2.0: August 14, 2010

- Add new methods to `QPDFObjectHandle` (`newStream` and `replaceStreamData` for creating new streams and replacing stream data. This makes it possible to perform a wide range of operations that were not previously possible.
- Add new helper method in `QPDFObjectHandle` (`addPageContents`) for appending or prepending new content streams to a page. This method makes it possible to manipulate content streams without having to be concerned whether a page's contents are a single stream or an array of streams.
- Add new method in `QPDFObjectHandle`: `replaceOrRemoveKey`, which replaces a dictionary key with a given value unless the value is null, in which case it removes the key instead.
- Add new method in `QPDFObjectHandle`: `getRawStreamData`, which returns the raw (unfiltered) stream data into a buffer. This complements the `getStreamData` method, which returns the filtered (uncompressed) stream data and can only be used when the stream's data is filterable.
- Provide two new examples: **`pdf-double-page-size`** and **`pdf-invert-images`** that illustrate the newly added interfaces.
- Fix a memory leak that would cause loss of a few bytes for every object involved in a cycle of object references. Thanks to Jian Ma for calling my attention to the leak.

2.1.5: April 25, 2010

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

2.1.4: April 18, 2010

- Apply the same padding calculation fix from version 2.1.2 to the main cross reference stream as well.
- Since **`qpdf --check`** only performs limited checks, clarify the output to make it clear that there still may be errors that qpdf can't check. This should make it less surprising to people when another PDF reader is unable to read a file that qpdf thinks is okay.

2.1.3: March 27, 2010

- Fix bug that could cause a failure when rewriting PDF files that contain object streams with unreferenced objects that in turn reference indirect scalars.
- Don't complain about (invalid) AES streams that aren't a multiple of 16 bytes. Instead, pad them before decrypting.

2.1.2: January 24, 2010

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

2.1.1: December 14, 2009

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

2.1: October 30, 2009

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

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

- Žarko Gajić has written a Delphi wrapper for qpdf, which can be downloaded from qpdf's download side. Žarko's Delphi wrapper is released with the same licensing terms as qpdf itself and comes with this disclaimer: "Delphi wrapper unit `qpdf.pas` created by Žarko Gajić (http://zarko-gajic.iz.hr/). Use at your own risk and for whatever purpose you want. No support is provided. Sample code is provided."
- Support has been added for AES encryption and crypt filters. Although qpdf does not presently support files that use PKI-based encryption, with the addition of AES and crypt filters, qpdf is now be able to open most encrypted files created with newer versions of Acrobat or other PDF creation software. Note that I have not been able to get very many files encrypted in this way, so it's possible there could still be some cases that qpdf can't handle. Please report them if you find them.
- Many error messages have been improved to include more information in hopes of making qpdf a more useful tool for PDF experts to use in manually recovering damaged PDF files.
- Attempt to avoid compressing metadata streams if possible. This is consistent with other PDF creation applications.
- Provide new command-line options for AES encrypt, cleartext metadata, and setting the minimum and forced PDF versions of output files.
- Add additional methods to the `QPDF` object for querying the document's permissions. Although qpdf does not enforce these permissions, it does make them available so that applications that use qpdf can enforce permissions.
- The *--check* option to **qpdf** has been extended to include some additional information.
- *Non-compatible API changes:*
  - QPDF's exception handling mechanism now uses `std::logic_error` for internal errors and `std::runtime_error` for runtime errors in favor of the now removed `QEXC` classes used in previous versions. The `QEXC` exception classes predated the addition of the `<stdexcept>` header file to the C++ standard library. Most of the exceptions thrown by the qpdf library itself are still of type `QPDFExc` which is now derived from `std::runtime_error`. Programs that catch an instance of `std::exception` and displayed it by calling the `what()` method will not need to be changed.
  - The `QPDFExc` class now internally represents various fields of the error condition and provides interfaces for querying them. Among the fields is a numeric error code that can help applications act differently on (a small number of) different error conditions. See `QPDFExc.hh` for details.
  - Warnings can be retrieved from qpdf as instances of `QPDFExc` instead of strings.
  - The nested `QPDF::EncryptionData` class's constructor takes an additional argument. This class is primarily intended to be used by `QPDFWriter`. There's not really anything useful an end-user application could do with it. It probably shouldn't really be part of the public interface to begin with. Likewise, some of the methods for computing internal encryption dictionary parameters have changed to support `/R=4` encryption.
  - The method `QPDF::getUserPassword` has been removed since it didn't do what people would think it did. There are now two new methods: `QPDF::getPaddedUserPassword` and `QPDF::getTrimmedUserPassword`. The first one does what the old `QPDF::getUserPassword` method used to do, which is to return the password with possible binary padding as specified by the PDF specification. The second one returns a human-readable password string.
  - The enumerated types that used to be nested in `QPDFWriter` have moved to top-level enumerated types and are now defined in the file `qpdf/Constants.h`. This enables them to be shared by both the C and C++ interfaces.

2.0.6: May 3, 2009

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

2.0.5: March 10, 2009

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

2.0.4: February 21, 2009

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

2.0.3: February 15, 2009

- Compile cleanly with gcc 4.4.
- Handle strings encoded as UTF-16BE properly.

2.0.2: June 30, 2008

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

2.0.1: May 6, 2008

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

2.0: April 29, 2008

- First public release.

CHAPTER

EIGHTEEN

ACKNOWLEDGMENTS

QPDF was originally created in 2001 and modified periodically between 2001 and 2005 during my employment at Apex CoVantage. Upon my departure from Apex, the company graciously allowed me to take ownership of the software and continue maintaining it 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.

In 2020, I joined Advent Health Partners, which has sponsored some previous QPDF work and generously allows me to spend some "company time" maintaining QPDF.

CHAPTER

NINETEEN

INDICES

- qpdf-options

a

--accessibility, 37
--add-attachment, 42
--allow-insecure, 38
--allow-weak-crypto, 24
--annotate, 37
--assemble, 37

c

--check, 45
--check-linearization, 45
--cleartext-metadata, 38
--coalesce-contents, 30
--collate, 32
--completion-bash, 23
--completion-zsh, 23
--compress-streams, 28
--compression-level, 29
--copy-attachments-from, 42
--copy-encryption, 27
--copyright, 23
--creationdate, 43

d

--decode-level, 28
--decrypt, 27
--description, 43
--deterministic-id, 24

e

--empty, 22
--encrypt, 27
--encryption-file-password, 27
--externalize-inline-images, 30
--extract, 37

f

--filename, 43
--filtered-stream-data, 45
--flatten-annotations, 33
--flatten-rotation, 33
--force-R5, 38
--force-V4, 38
--force-version, 31
--form, 37
--from, 41

g

--generate-appearances, 34

h

--help, 23

i

--ignore-xref-streams, 26
--ii-min-bytes, 30
--is-encrypted, 44

j

--job-json-file, 22
--job-json-help, 23
--json, 46
--json-help, 47
--json-input, 47
--json-key, 47
--json-object, 47
--json-output, 47
--json-stream-data, 47
--json-stream-prefix, 47

k

--keep-files-open, 25
--keep-files-open-threshold, 25
--keep-inline-images, 35
--key, 43

l

--linearize, 27
--linearize-pass1, 48
--list-attachments, 46

m

--mimetype, 43
--min-version, 30

--moddate, 43
--modify, 37
--modify-other, 37

n

--newline-before-endstream, 30
--no-original-object-ids, 28
--no-warn, 24
--normalize-content, 29

o

--object-streams, 29
--oi-min-area, 35
--oi-min-height, 35
--oi-min-width, 35
--optimize-images, 34
--overlay, 33

p

--pages, 32
--password, 24
--password-file, 24
--password-is-hex-key, 25
--password-mode, 25
--prefix, 44
--preserve-unreferenced, 29
--preserve-unreferenced-resources, 30
--print, 38
--progress, 24

q

--qdf, 27

r

--raw-stream-data, 45
--recompress-flate, 29
--remove-attachment, 42
--remove-page-labels, 35
--remove-unreferenced-resources, 30
--repeat, 41
--replace, 43
--replace-input, 22
--report-memory-usage, 48
--requires-password, 44
--rotate, 33

s

--show-attachment, 46
--show-crypto, 23
--show-encryption, 45
--show-encryption-key, 45
--show-linearization, 45
--show-npages, 46
--show-object, 45
--show-pages, 46
--show-xref, 45
--split-pages, 32
--static-aes-iv, 48
--static-id, 48
--stream-data, 28
--suppress-password-recovery, 25
--suppress-recovery, 26

t

--test-json-schema, 48
--to, 41

u

--underlay, 33
--update-from-json, 47
--use-aes, 38

v

--verbose, 24
--version, 23

w

--warning-exit-0, 22
--with-images, 46