its ***PointerHolder<QDFObject>*** with the one from the newly returned ***QPDFObjectHandle***. In this way, only a single copy of any direct object need exist and clients can access objects transparently without knowing caring whether they are direct or indirect objects. Additionally, no object is ever read from the file more than once. That means that only the portions of the PDF file that are actually needed are ever read from the input file, thus allowing the qpdf package to take advantage of this important design goal of PDF files.

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

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

- Client constructs ***QPDF*** *pdf* and calls *pdf.processFile("a.pdf");*.
- The ***QPDF*** class checks the beginning of *a.pdf* for a PDF header. It then reads the cross reference table mentioned at the end of the file, ensuring that it is looking before the last `%%EOF`. After getting to `trailer` keyword, it invokes the parser.
- The parser sees "`<<`", so it calls itself recursively in dictionary creation mode.
- In dictionary creation mode, the parser keeps accumulating objects until it encounters "`>>`". Each object that is read is pushed onto a stack. If "`R`" is read, the last two objects on the stack are inspected. If they are integers, they are popped off the stack and their values are used to construct an indirect object handle which is then pushed onto the stack. When "`>>`" is finally read, the stack is converted into a ***QPDF_Dictionary*** which is placed in a ***QPDFObjectHandle*** and returned.
- The resulting dictionary is saved as the trailer dictionary.
- The `/Prev` key is searched. If present, ***QPDF*** seeks to that point and repeats except that the new trailer dictionary is not saved. If `/Prev` is not present, the initial parsing process is complete.

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

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

7.5. Casting Policy

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

The C++ code in qpdf is free of old-style casts except where unavoidable (e.g. where the old-style cast is in a macro provided by a third-party header file). When there is a need for a cast, it is handled, in order of preference, by rewriting the code to avoid the need for a cast, calling *const_cast*, calling *static_cast*, calling *reinterpret_cast*, or calling some