MASTERING PDF PARSING

BUILDING A PDF READER FROM THE ISO 32000-2 SPECIFICATION

- Understand the PDF 2.0 standard in technical depth
- Implement a complete, standards-compliant PDF parser in C++
- From file structure to rendering primitives and beyond
- Open, validate, navigate, and extract information from real-world PDFs

FELIX HARTMANN

C++

Mastering PDF Parsing

Building a PDF Reader from the ISO 32000-2 Specification

Steve Publications

This book is available at https://leanpub.com/masteringpdfparsing

This version was published on 2026-07-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Building a PDF Reader from the ISO 32000-2 Specification

This book teaches you how the PDF file format works by progressively designing and implementing a complete, standards-compliant PDF reader and parser from scratch based on the latest ISO 32000-2 specification. Beginning with the history and architecture of PDF, each chapter explains a core component of the format in technical depth while extending a working C++ implementation. By the final chapter, you will have built a fully functional PDF parser capable of opening, validating, navigating, and extracting information from real-world PDF documents without relying on any third-party PDF libraries. The code is production-quality, the explanations are rigorous, and every claim traces back to the specification itself. This is both a deep exploration of the PDF 2.0 standard and a complete implementation guide for experienced software developers.

Introduction: Why Build a PDF Parser?

Open any PDF file in a hex editor, and you will see something that looks like structured text mixed with binary gibberish. Lines beginning with percent signs declare the version. Numbers followed by the word "obj" mark the start of objects. Curly braces contain dictionaries. The letters "stream" and "endstream" bracket compressed data. At the bottom of the file, a cross-reference table maps object numbers to byte offsets, and a trailer dictionary points back to the document's root. This is not a random arrangement. Every character has a purpose defined in ISO 32000-2, a specification of nearly one thousand pages that governs how PDF documents are structured, encoded, and interpreted.

Yet for all its ubiquity, PDF remains one of the most complex file formats in widespread use. It is a format built over thirty years of incremental additions, each layer designed to solve a specific problem while maintaining backward compatibility with everything that came before. The result is a specification that reads like a geological record: PostScript-derived graphics operators layered beneath object streams and cross-reference streams, transparency models grafted onto a fundamentally opaque rendering pipeline, XML Forms Architecture nested inside an otherwise binary format only to be deprecated later. Understanding PDF requires navigating not just its current state but the historical decisions that shaped it.

This book takes a different approach than a specification reference or a library tutorial. It teaches you how PDF works by making you build a parser from the ground up. We will read ISO 32000-2 section by section, translate each requirement into working C++ code, and test the implementation against real PDF documents. By the end, you will have a complete PDF reader that can open any conforming document, navigate its structure, extract text and metadata, and validate compliance with the standard. More importantly, you will understand why the format works the way it does and what happens when it goes wrong.

The Ubiquity and Hidden Complexity of PDF

PDF is everywhere. It is the de facto standard for document exchange across industries: legal contracts, academic papers, government forms, technical manuals, financial reports. When someone says "send me a PDF," they mean a file that will look the same on every device, printer, and operating system. This portability is not accidental. It is the direct result of design decisions made in 1991 by John Warnock at Adobe Systems, who envisioned a format you could "capture from any application, send anywhere, and view and print on any machine."

The hidden complexity lies beneath this simplicity. A PDF file is not a document in the way a word processor thinks of documents. It is a collection of objects organized into a graph: pages reference resources, resources reference fonts and images, fonts reference character maps, annotations reference actions, actions reference destinations. Objects can be direct (embedded inline) or indirect (referenced by number and generation). Streams can be compressed with multiple filters applied in sequence. Cross-reference tables can be replaced by cross-reference streams. The same visual document can be represented by vastly different internal structures depending on which tool created it.

This complexity creates real challenges for anyone building PDF software. Consider these problems that any robust parser must handle:

- **Malformed documents**: Real-world PDFs frequently violate the specification in ways that still render correctly in major viewers. A parser that rejects every non-conforming file is useless. A parser that accepts everything is insecure. The balance between strictness and tolerance is the central tension of PDF parsing.
- **Incremental updates**: PDF supports appending new content to existing files without rewriting the entire document. This means a single file can contain multiple cross-reference tables and trailers, each describing a different version of the document. The parser must follow the chain to find the latest state.
- **Object streams**: Introduced in PDF 1.5, object streams compress multiple objects into a single stream. Objects inside are not directly addressable by byte offset; they must be extracted from the decompressed stream data first.

- **Encryption**: PDF encryption is notoriously complex, with multiple revisions of key derivation algorithms, password handling schemes, and cipher modes. The standard security handler alone spans dozens of pages in the specification.
- **Fonts and text extraction**: Extracting readable text from a PDF is harder than it sounds. Characters in content streams are encoded as byte sequences that must be mapped through font encodings, CMaps, and ToUnicode tables to produce Unicode strings. Different fonts on the same page may use completely different encoding schemes.

These are not edge cases. They are the daily reality of working with PDF documents at scale. Building a parser that handles them correctly requires understanding the format at a level most developers never need. This book provides that understanding.

What This Book Will Build

The implementation we will build is a C++17 library called PdfReader (the name is intentionally generic). It is designed to be:

- **Standards-compliant**: Every parsing decision traces back to ISO 32000-2. Where the specification allows discretion, we follow the principle of being liberal in reading and strict in writing.
- **Incremental**: The library grows chapter by chapter. Each addition is backward compatible with previous code. You can compile and test the implementation after every chapter.
- **Production-quality**: Error handling is explicit. Memory management is safe (using RAII and smart pointers). Performance considerations are addressed where they matter.
- **Self-contained**: No third-party PDF libraries are used. External dependencies are limited to standard C++17 and zlib for FlateDecode compression.

By the end of the book, PdfReader will support:

- Opening and parsing any PDF version from 1.0 through 2.0
- Reading both traditional cross-reference tables and cross-reference streams

- Decompressing all standard filters (FlateDecode, LZWDecode, ASCIIHexDecode, ASCII85Decode, RunLengthDecode)
- Navigating the document catalog and page tree
- Parsing content streams and understanding graphics operators
- Extracting text with proper font encoding and CMap handling
- Reading metadata (both legacy Info dictionaries and XMP packets)
- Processing annotations, links, and destinations
- Handling AcroForm fields and interactive forms
- Validating digital signatures
- Detecting and recovering from malformed documents

What PdfReader will not do is render pages to pixels. Rendering is a separate domain requiring a graphics engine, font rasterizer, and compositing pipeline. Our focus is parsing: reading the file format, building an in-memory representation of the document structure, and extracting information. If you want to render PDFs, you can build on top of this parser or use a dedicated rendering library.

How to Read This Book: Code Structure and Conventions

This book assumes you are an experienced software developer comfortable with systems programming concepts: pointers, memory management, file I/O, data structures, and algorithm design. The code is written in C++17 because it gives us the control we need over low-level operations while providing modern safety features. If your primary language is something else, the concepts transfer directly.

The implementation is organized into headers and source files that mirror the logical structure of the PDF format:

- `pdf_types.h`: Forward declarations for core PDF object types
- `pdf_file.h` / `pdf_file.cpp`: File I/O abstraction with random access, including the `PdfReadSource` interface and `MemoryBuffer` for in-memory parsing
- `pdf_lex.h` / `pdf_lex.cpp`: Lexical analysis and tokenization
- `pdf_parse.h` / `pdf_parse.cpp`: Recursive-descent parser for PDF objects

- `pdf_object.h` / `pdf_object.cpp`: Object model (direct and indirect), including all PDF object types
- `pdf_context.h` / `pdf_context.cpp`: Runtime context managing the object cache, file handle, and decryptor
- `pdf_xref.h` / `pdf_xref.cpp`: Cross-reference table and stream handling
- `pdf_filters.h` / `pdf_filters.cpp`: Stream decompression filters (ASCIIHex, ASCII85, LZW, Flate, RunLength) with PNG predictor support
- `pdf_stream.h` / `pdf_stream.cpp`: Stream data loading with decryption and filter chain application
- `pdf_document.h` / `pdf_document.cpp`: Document catalog and navigation
- `pdf_page.h` / `pdf_page.cpp`: Page tree and page objects
- `pdf_font.h` / `pdf_font.cpp`: Font dictionaries, ToUnicode CMap parsing, text encoding helpers
- `pdf_content.h` / `pdf_content.cpp`: Content stream parsing and text extraction
- `pdf_annotation.h` / `pdf_annotation.cpp`: Annotations and actions
- `pdf_form.h` / `pdf_form.cpp`: AcroForm fields
- `pdf_security.h` / `pdf_security.cpp`: Encryption (R2–R6) and digital signatures
- `pdf_metadata.h` / `pdf_metadata.cpp`: Metadata extraction

Each chapter introduces new files and extends existing ones. Code listings show complete, compilable implementations. When a function spans many lines, the full implementation is shown, not abbreviated. You can copy the code directly and compile it.

Some conventions used throughout:

- ISO 32000-2 section numbers are cited inline whenever we discuss a specific requirement (e.g., “per ISO 32000-2 Section 7.2.3”).
- Byte offsets and hexadecimal values use the suffix “h” to match the specification’s notation (e.g., 0Ah for line feed).
- PDF names are written with the leading slash as they appear in files (e.g., /Type, /Pages).
- Error handling uses exceptions for unrecoverable conditions and returns optional values or error codes for expected failures.

- All code is tested against real PDF documents from the Mozilla pdf.js test suite and the Corkami PDF corpus.

Key architectural patterns:

- **`PdfReadSource` abstraction**: Both file-based and memory-based parsing share the same interface, enabling the lexer and parser to work with data from any source.
- **`PdfContext` as central runtime state**: A single context object holds the file handle, parsed object cache, and decryptor, passed through the parsing pipeline to avoid global state.
- **Object-oriented type hierarchy**: All PDF objects derive from `PdfObject`, with polymorphic `type_name()` and `to_string()` methods for debugging and serialization.
- **RAII and smart pointers**: Resources are managed automatically; indirect objects are stored as `std::shared_ptr<PdfObject>` (`ObjectPtr`) to handle shared references safely.

Getting Started: Project Setup and Toolchain

To follow along with the implementation, you will need:

- A C++17-compatible compiler (GCC 8+, Clang 7+, or MSVC 2019+)
- CMake 3.16+ for build configuration
- zlib development headers for FlateDecode support
- A hex editor (such as HxD, Bless, or xxd) for inspecting PDF files
- Several sample PDF documents for testing

Create a project directory with the following structure:

```
pdfreader/
  CMakeLists.txt
  include/
    pdf_types.h         # Forward declarations
    pdf_file.h          # PdfFile, MemoryBuffer, PdfReadSource
    pdf_lex.h           # Lexer, Token, TokenType
    pdf_parse.h         # PdfParser
    pdf_object.h        # All PDF object types
    pdf_context.h       # PdfContext (object cache, file, decryptor)
    pdf_xref.h          # XrefTable, XrefEntry
    pdf_filters.h       # PdfFilters (all decompression algorithms)
    pdf_stream.h        # load_stream_data()
    pdf_document.h      # PdfDocument
    pdf_page.h          # PdfPage
    pdf_font.h          # ToUnicode CMap parsing, font helpers
    pdf_content.h       # ContentStreamParser, extract_text()
    pdf_annotation.h    # PdfAnnotation, extract_annotations()
    pdf_form.h          # FormField, extract_form_fields()
    pdf_security.h      # PdfDecryptor, PdfSignature
    pdf_metadata.h      # PdfMetadata, extract_metadata()
  src/
    pdf_file.cpp
    pdf_lex.cpp
    pdf_parse.cpp
    pdf_object.cpp
    pdf_context.cpp
    pdf_xref.cpp
    pdf_filters.cpp
    pdf_stream.cpp
    pdf_document.cpp
    pdf_page.cpp
    pdf_font.cpp
    pdf_content.cpp
    pdf_annotation.cpp
    pdf_form.cpp
    pdf_security.cpp
    pdf_metadata.cpp
  tests/
    test_reader.cpp
  samples/
    sample.pdf          # Sample PDF for testing
```

The CMakeLists.txt file:

```
cmake_minimum_required(VERSION 3.16)
project(pdfreader VERSION 1.0.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(ZLIB REQUIRED)

add_library(pdfreader
    src/pdf_file.cpp
    src/pdf_lex.cpp
    src/pdf_parse.cpp
    src/pdf_object.cpp
    src/pdf_xref.cpp
    src/pdf_stream.cpp
    src/pdf_document.cpp
    src/pdf_page.cpp
    src/pdf_font.cpp
    src/pdf_content.cpp
    src/pdf_annotation.cpp
    src/pdf_form.cpp
    src/pdf_security.cpp
    src/pdf_metadata.cpp
    src/pdf_context.cpp
    src/pdf_filters.cpp
)

target_include_directories(pdfreader PUBLIC include)
target_link_libraries(pdfreader PUBLIC ZLIB::ZLIB)

enable_testing()
add_executable(test_reader tests/test_reader.cpp)
target_link_libraries(test_reader pdfreader)
add_test(NAME reader_tests COMMAND test_reader)
```

Build with:

```
mkdir build && cd build
cmake ..
make
ctest
```

The first chapter begins the journey by examining the history and architecture of PDF, then implements the file I/O layer that all subsequent parsing depends on. From there, we move through lexical analysis, object parsing, cross-reference handling, stream decoding, and everything else required to build a complete PDF reader. Let us begin.

Chapter 1: History, Design Goals, and Architecture of PDF

PostScript Heritage and the Birth of Acrobat

To understand why PDF works the way it does, you need to understand PostScript. In the early 1980s, printing was a painful process. Different printers used different control languages. Fonts had to be physically installed in the printer hardware. What looked correct on screen rarely matched what came out of the printer. Adobe Systems, founded in 1982 by John Warnock and Charles Geschke, set out to solve this problem with PostScript: a page description language based on the Forth programming language that could describe any printable page as a sequence of drawing commands.

PostScript was revolutionary. It treated the printed page as the primary output device and described everything in terms of vector graphics, text placement, and color. A PostScript program could draw lines, curves, and rectangles, position text using embedded font outlines, and compose complex layouts with mathematical precision. The language was Turing-complete, meaning it could express any computation needed to generate a page. By the late 1980s, PostScript had become the standard for professional printing.

But PostScript had a fatal flaw for document exchange: it was a programming language, not a document format. A PostScript file described how to draw a page, but the execution order and timing depended on the interpreter. Pages could be rendered in any order. Random access to individual pages was difficult. The language's full programmability meant that malicious or buggy PostScript files could consume unlimited resources or even crash the interpreter. For reliable document viewing across different systems, a simpler, more constrained format was needed.

Development of PDF began in 1991 when John Warnock wrote a paper for a project code-named Camelot. The goal was to create a simplified version of PostScript called Interchange PostScript (IPS) that would preserve the visual

appearance of documents without the complexity and risks of full PostScript programming. Key design decisions distinguished PDF from its ancestor:

- **Pre-computed layout**: Unlike PostScript, where pages are generated by executing a program, PDF stores the final result of all computations. Text positions, font metrics, and graphic transformations are fixed values, not expressions to be evaluated.
- **Object-based structure**: PDF represents documents as collections of objects (strings, numbers, arrays, dictionaries, streams) rather than a linear sequence of commands. Objects can reference each other, enabling shared resources and random access.
- **Deterministic rendering**: A PDF viewer does not execute code; it interprets a fixed set of operators in a well-defined order. This eliminates the non-determinism and security risks of PostScript execution.
- **Binary efficiency**: While PostScript is purely textual, PDF allows binary data for streams (images, compressed content), reducing file sizes significantly.

Version 1.0 of PDF was announced at Comdex Fall in 1992, where it won a "best of Comdex" award. Adobe Acrobat 1.0 was released on June 15, 1993, comprising three components: Acrobat Reader (the free viewer), Acrobat Exchange (the editing and creation tool), and Acrobat Distiller (which converted PostScript files to PDF). Early adoption was slow. File sizes were large, hyperlinks were not yet supported, and the cost of creation tools was prohibitive. The Adobe Board of Directors considered canceling the project. What saved PDF was the decision to make Acrobat Reader free starting with version 2.0, which dramatically expanded the installed base and made PDF a practical format for document distribution.

Design Principles: Device Independence, Portability, Interactivity

The design principles that guided PDF's creation continue to influence the format today. ISO 32000-2 Section 0.1 states the core requirements that PDF fulfills:

1. **Device independence**: A PDF document shall look the same on any output device, whether a high-resolution monitor, a laser printer, or a mobile

phone screen. This is achieved through the PDF Imaging Model, which describes all visual content in terms of abstract operations (draw this path, fill with this color, show this text at this position) that are mapped to the target device's capabilities at render time.

2. **Portability**: A PDF document shall be viewable and printable without requiring the original authoring application or any specific fonts. This is achieved through embedded font programs and self-contained resource dictionaries. Every page carries with it everything needed to render itself: the fonts, images, color profiles, and content streams.
3. **Interactivity**: A PDF document may contain interactive features such as hyperlinks, form fields, annotations, and multimedia content. These features are implemented through dedicated object types (annotation dictionaries, action dictionaries, field dictionaries) that are separate from the page content itself.
4. **Efficiency**: A PDF file shall be compact enough for practical distribution over networks and storage on limited-capacity devices. This is achieved through compression filters, shared resources, and the ability to store binary data directly rather than encoding everything as text.

These principles explain many of PDF's architectural choices. The object model exists to enable resource sharing and random access. The separation of content streams from annotations allows interactive features to be added without modifying the underlying page graphics. The filter system enables compression without changing the logical structure of the document. Understanding these principles helps explain not just what PDF does but why it does it that way.

Version Timeline from PDF 1.0 to PDF 2.0

PDF has evolved through numerous versions, each adding features while maintaining backward compatibility. A conforming PDF processor for version N must be able to read any document created with version M where M <= N. This requirement has led to a specification that accumulates features over time without removing old ones (until PDF 2.0, which finally deprecated some legacy features).

Here is the version timeline with key additions:

- **PDF 1.0 (1993)**: The original format. Basic text, vector graphics, and raster images. PostScript-derived content stream operators. Simple encryption (40-bit RC4 introduced later as a patch).
- **PDF 1.1 (1996)**: Added support for embedded Type 1 fonts, incremental file updates, and the ability to specify page labels.
- **PDF 1.2 (1996)**: Introduced TrueType font embedding, thumbnail images, and linearization (Fast Web View) for progressive rendering over slow networks.
- **PDF 1.3 (1999)**: Added JPEG 2000 compression, transparency groups, and the ability to embed OpenType fonts. Associated with Acrobat 4.0.
- **PDF 1.4 (2001)**: Major release introducing the transparency model (constant alpha, soft masks, blend modes), layers (optional content groups), and file attachments. Associated with Acrobat 5.0.
- **PDF 1.5 (2003)**: Introduced object streams for compressing multiple objects together, cross-reference streams as a compressed alternative to traditional xref tables, and incremental updates improvements. Associated with Acrobat 6.0.
- **PDF 1.6 (2005)**: Added 3D content support (U3D and PRC formats), structured document storage for embedded files, and improved security with AES-128 encryption. Associated with Acrobat 7.0.
- **PDF 1.7 (2006)**: The most comprehensive release, adding digital signatures, redaction annotations, video support, and many refinements. Published as ISO 32000-1:2008 in 2008. Associated with Acrobat 8.0.
- **PDF 2.0 (ISO 32000-2:2017)**: First version developed entirely within ISO rather than by Adobe. Removed all proprietary extension levels, deprecated XFA forms, improved transparency handling, added AES-256 encryption as the recommended algorithm, and clarified many ambiguities in earlier versions. Revised in ISO 32000-2:2020 with corrections.

Between PDF 1.7 and PDF 2.0, Adobe released proprietary "Extension Levels" (1 through 8) that added features beyond the ISO standard. These included XFA dynamic forms, enhanced digital signatures, and improved color management. PDF 2.0 incorporated many of these features into the base specification while removing others, creating a clean break from the extension-level model.

The version number appears in the file header as a comment line, for example:

```
%PDF-1.7
```

This is not merely metadata. It tells the parser which features to expect. A document claiming to be PDF 1.4 cannot use object streams (a PDF 1.5 feature), and a parser reading a PDF 2.0 document knows that deprecated features like XFA should be ignored. In practice, many PDF generators set the version incorrectly (either too low or too high), so a robust parser must validate features against actual content, not just the declared version.

The Object Model: A Foundation for Everything

At the core of PDF is the object model. Every piece of data in a PDF file is an object. Objects are of nine types, defined in ISO 32000-2 Section 7.3:

1. **Null**: A single value representing the absence of a value, written as the keyword `null`.
2. **Boolean**: The values `true` and `false`, written as keywords.
3. **Integer**: Whole numbers in the range -2^{31} to $2^{31}-1$, written as sequences of decimal digits optionally preceded by a minus sign.
4. **Real**: Floating-point numbers, written as decimal digits with a period separating the integer and fractional parts, optionally preceded by a minus sign.
5. **String**: Sequences of bytes, either literal (enclosed in parentheses) or hexadecimal (enclosed in angle brackets).
6. **Name**: Identifiers beginning with a slash, used as dictionary keys and symbolic constants.
7. **Array**: Ordered sequences of objects enclosed in square brackets.
8. **Dictionary**: Associative arrays of name-value pairs enclosed in double angle brackets.
9. **Stream**: A dictionary followed by a sequence of bytes, used for large data such as images and compressed content.

Objects can be direct or indirect. Direct objects appear inline within other objects. Indirect objects are assigned a unique object number and generation number and can be referenced from anywhere in the file using an indirect reference (two numbers followed by R, such as `3 0 R`). The distinction matters: indirect objects can be updated independently through incremental updates,

and they enable multiple parts of the document to share the same resource without duplication.

The entire document structure is built from these objects. The catalog dictionary references the pages object. Pages objects reference individual page objects. Page objects reference content streams, fonts, and images. Annotations reference actions and destinations. Every relationship is an object reference. This uniformity is one of PDF's greatest strengths: the same parsing logic handles every part of the file, regardless of its semantic role.

File Structure at a Glance: Headers, Bodies, Cross-References, Trailers

A PDF file has four logical sections, as shown in ISO 32000-2 Figure 5:

```
+------------------+
|     Header       |   %PDF-x.x
+------------------+
|     Body         |   Objects (direct and indirect)
+------------------+
| Cross-Reference  |   xref table or xref stream
+------------------+
|    Trailer       |   trailer dictionary + startxref + %%EOF
+------------------+
```

The **header** is a single line beginning with `%PDF-` followed by the version number. It must be the first bytes of the file. After the header, binary data may appear immediately, so parsers should not assume the file is purely ASCII.

The **body** contains all the document's objects, written sequentially. Objects in the body are not necessarily in numerical order, and they are not padded to fixed boundaries. The only way to locate an indirect object efficiently is through the cross-reference table.

The **cross-reference section** provides a mapping from object numbers to byte offsets within the file. Traditional cross-reference tables (present since PDF 1.0) use a fixed-format text representation with one line per object. Cross-reference streams (introduced in PDF 1.5) compress this data into a binary stream, significantly reducing file size for large documents.

The **trailer** is a dictionary that provides entry points into the document structure. It contains at minimum:

- `/Root`: An indirect reference to the catalog dictionary (the root of the document object tree).
- `/Size`: The total number of entries in the cross-reference table.
- `/Prev` (optional): The byte offset of a previous cross-reference section, used for incremental updates.

The trailer is followed by the keyword `startxref`, the byte offset of the cross-reference section, and the end-of-file marker `%%EOF`. This arrangement allows a parser to read the file from the end: find `%%EOF`, back up to `startxref`, read the cross-reference offset, jump to the cross-reference table, and then navigate to any object by its byte offset.

Building the File I/O Layer

Before we can parse anything, we need a reliable way to read PDF data. PDF parsing requires random access: jumping to arbitrary byte offsets to read objects referenced by the cross-reference table. Sequential reading alone is insufficient. Our file I/O layer must support:

1. Opening a file and determining its size
2. Reading bytes at any offset
3. Efficiently seeking to arbitrary positions
4. Detecting and reporting I/O errors

But there is a subtlety: we also need to parse data from memory buffers. When we extract objects from object streams (PDF 1.5+), those objects are stored in decompressed byte arrays, not on disk. We want the same lexer and parser to work for both file-based and memory-based input. To achieve this, we define a common interface called `PdfReadSource` that abstracts the source of bytes:

```cpp
// include/pdf_file.h
#ifndef PDF_FILE_H
#define PDF_FILE_H

#include <cstddef>
#include <cstdint>
#include <cstring>
#include <fstream>
#include <optional>
#include <string>
#include <system_error>
#include <vector>

namespace pdf {

// Common interface for reading PDF data (from file or memory).
class PdfReadSource {
public:
    virtual ~PdfReadSource() = default;

    // Get the total size in bytes.
    virtual std::size_t size() const = 0;

    // Read exactly 'count' bytes starting at 'offset'.
    // Returns false if the read would extend past EOF or on I/O error.
    virtual bool read_at(std::size_t offset, void* buffer,
                         std::size_t count) const = 0;

    // Read exactly 'count' bytes starting at 'offset' into a vector.
    // Returns nullopt if the read would extend past EOF or on I/O error.
    virtual std::optional<std::vector<uint8_t>> read_at(
        std::size_t offset, std::size_t count) const = 0;

    // Read a null-terminated string at 'offset', up to max_length bytes.
    virtual std::optional<std::string> read_string_at(
        std::size_t offset, std::size_t max_length) const = 0;
};

// In-memory buffer that provides the PdfReadSource interface.
// Used for parsing data from streams (e.g., object streams).
class MemoryBuffer : public PdfReadSource {
public:
    explicit MemoryBuffer(const std::vector<uint8_t>& data)
        : data_(data) {}

    // Non-copyable, movable
    MemoryBuffer(const MemoryBuffer&) = delete;
    MemoryBuffer& operator=(const MemoryBuffer&) = delete;
    MemoryBuffer(MemoryBuffer&&) noexcept = default;
    MemoryBuffer& operator=(MemoryBuffer&&) noexcept = default;
```

```
    ~MemoryBuffer() override = default;

    std::size_t size() const override { return data_.size(); }

    bool read_at(std::size_t offset, void* buffer,
                 std::size_t count) const override {
        if (offset + count > data_.size()) return false;
        std::memcpy(buffer, data_.data() + offset, count);
        return true;
    }

    std::optional<std::vector<uint8_t>> read_at(
        std::size_t offset, std::size_t count) const override {
        if (offset + count > data_.size()) return std::nullopt;
        return std::vector<uint8_t>(data_.begin() + offset,
                                    data_.begin() + offset + count);
    }

    std::optional<std::string> read_string_at(
        std::size_t offset, std::size_t max_length) const override {
        if (offset >= data_.size()) return std::nullopt;
        std::size_t end = offset;
        std::size_t limit = std::min(offset + max_length, data_.size());
        while (end < limit && data_[end] != 0) ++end;
       return std::string(reinterpret_cast<const char*>(data_.data() + offset),
                          end - offset);
    }

private:
    std::vector<uint8_t> data_;
};

class PdfFile : public PdfReadSource {
public:
    // Open a PDF file for reading. Returns std::error_code on failure.
    static std::optional<PdfFile> open(const std::string& path);

    // Non-copyable, movable
    PdfFile(const PdfFile&) = delete;
    PdfFile& operator=(const PdfFile&) = delete;
    PdfFile(PdfFile&&) noexcept = default;
    PdfFile& operator=(PdfFile&&) noexcept = default;
    ~PdfFile() override = default;

    // Get the total size of the file in bytes.
    std::size_t size() const override;

    // Read exactly 'count' bytes starting at 'offset'.
    // Returns false if the read would extend past EOF or on I/O error.
    bool read_at(std::size_t offset, void* buffer,
```

```
                    std::size_t count) const override;

    // Read exactly 'count' bytes starting at 'offset' into a vector.
    // Returns nullopt if the read would extend past EOF or on I/O error.
    std::optional<std::vector<uint8_t>> read_at(
        std::size_t offset, std::size_t count) const override;

    // Read a null-terminated string at 'offset', up to max_length bytes.
    std::optional<std::string> read_string_at(
        std::size_t offset, std::size_t max_length) const override;

    // Get the underlying file stream (for advanced operations).
    const std::ifstream& stream() const;

private:
    PdfFile(std::ifstream file, std::size_t file_size);

    std::ifstream file_;
    std::size_t size_;
};

} // namespace pdf

#endif // PDF_FILE_H
```

The `PdfReadSource` abstraction is a key design decision. By defining a common interface for reading bytes, we can write our lexer and parser to work with any data source – files on disk, decompressed stream buffers, or even network streams in a future extension. This follows the dependency inversion principle: high-level parsing logic depends on an abstraction (`PdfReadSource`), not on concrete file I/O details.

`MemoryBuffer` wraps a `std::vector<uint8_t>` and implements the same interface as `PdfFile`. When we extract objects from object streams later in the book, we will wrap those decompressed buffers in `MemoryBuffer` and feed them directly to the parser, reusing all of our parsing logic without modification.

The `PdfFile` implementation handles error cases explicitly and uses RAII for resource management. Note the use of `const_cast` in the read methods: file operations like seeking and reading are logically const (they do not modify the file's conceptual state), but `std::ifstream` does not mark them as const. We use `mutable_cast` to satisfy the type system while preserving logical constness:

```cpp
// src/pdf_file.cpp
#include "pdf_file.h"
#include <algorithm>
#include <cstring>
#include <iostream>

namespace pdf {

std::optional<PdfFile> PdfFile::open(const std::string& path) {
    std::ifstream file(path, std::ios::binary | std::ios::ate);
    if (!file.is_open()) {
        return std::nullopt;
    }

    // Get file size by seeking to end
    auto end_pos = file.tellg();
    if (end_pos == std::streampos(-1)) {
        return std::nullopt;
    }

    std::size_t file_size = static_cast<std::size_t>(end_pos);

    // Seek back to beginning
    file.seekg(0, std::ios::beg);
    if (file.fail()) {
        return std::nullopt;
    }

    return PdfFile(std::move(file), file_size);
}

PdfFile::PdfFile(std::ifstream file, std::size_t file_size)
    : file_(std::move(file)), size_(file_size) {}

std::size_t PdfFile::size() const {
    return size_;
}

bool PdfFile::read_at(std::size_t offset, void* buffer,
                      std::size_t count) const {
    // Check bounds
    if (offset + count > size_) {
        return false;
    }

    // Use mutable cast for seeking/reading (file operations are logically
    ↪  const)
    auto& mutable_file = const_cast<std::ifstream&>(file_);

    // Seek to offset
```

```
    mutable_file.seekg(static_cast<std::streamoff>(offset), std::ios::beg);
    if (mutable_file.fail()) {
        return false;
    }

    // Read data
    mutable_file.read(static_cast<char*>(buffer),
              static_cast<std::streamsize>(count));
    return mutable_file.gcount() == static_cast<std::streamsize>(count);
}

std::optional<std::vector<uint8_t>> PdfFile::read_at(
    std::size_t offset, std::size_t count) const {
    if (offset + count > size_) {
        return std::nullopt;
    }

    std::vector<uint8_t> buffer(count);
    if (!read_at(offset, buffer.data(), count)) {
        return std::nullopt;
    }

    return buffer;
}

std::optional<std::string> PdfFile::read_string_at(
    std::size_t offset, std::size_t max_length) const {
    if (offset >= size_) {
        return std::nullopt;
    }

    // Read up to max_length bytes or until null terminator
    std::vector<uint8_t> buffer(max_length);
    auto& mutable_file = const_cast<std::ifstream&>(file_);
    mutable_file.seekg(static_cast<std::streamoff>(offset), std::ios::beg);
    if (mutable_file.fail()) {
        return std::nullopt;
    }

    mutable_file.read(reinterpret_cast<char*>(buffer.data()),
              static_cast<std::streamsize>(max_length));
    std::streamsize bytes_read = mutable_file.gcount();

    // Find null terminator
    std::size_t null_pos = 0;
    for (; null_pos < static_cast<std::size_t>(bytes_read); ++null_pos) {
        if (buffer[null_pos] == 0) {
            break;
        }
    }
```

```
    return std::string(reinterpret_cast<const char*>(buffer.data()), null_pos);
}

const std::ifstream& PdfFile::stream() const {
    return file_;
}

} // namespace pdf
```

This file I/O layer provides the foundation for everything that follows. The `read_at` method enables random access to any part of the file, which is essential for implementing cross-reference-based object lookup. The bounds checking prevents reading past the end of the file, a common source of bugs in PDF parsers.

Let us verify the implementation works with a simple test:

```
// tests/test_file.cpp
#include "pdf_file.h"
#include <cassert>
#include <iostream>

int main() {
    auto file = pdf::PdfFile::open("samples/hello.pdf");
    if (!file) {
        std::cerr << "Failed to open sample PDF\n";
        return 1;
    }

    std::cout << "File size: " << file->size() << " bytes\n";

    // Read header
    auto header = file->read_at(0, 8);
    if (header) {
        std::string hdr(header->begin(), header->end());
        std::cout << "Header: " << hdr << "\n";
    }

    return 0;
}
```

With the file I/O layer in place, we can now move to the next step: converting raw bytes into meaningful tokens. The next chapter implements the lexer that forms the foundation of PDF parsing.

Chapter 2: Lexical Structure and the Tokenizer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The PDF Character Set and Whitespace Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Delimiters, Brackets, and Parentheses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Numeric Tokens: Integers and Real Numbers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Names, Literal Strings, Hexadecimal Strings, Booleans, Null

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Comments and Escaping Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Building the Tokenizer: Complete Implementation

Chapter 3: Direct Objects and Recursive-Descent Parsing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The Parser Architecture: From Tokens to Object Trees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Parsing Numbers and Simple Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Parsing Literals Strings with Escape Sequences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Parsing Hexadecimal Strings and Length Validation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Parsing Names and Key Resolution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Parsing Dictionaries: Keys, Values, and Nested Structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Parsing Arrays and Recursive Descent Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Putting It All Together: A Simple Test

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 4: Indirect Objects, Object Numbering, and Streams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Why Indirect Objects Matter: Referencing and Random Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The obj-endobj Syntax and Object Numbers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Stream Objects: Headers, Lengths, and Data Boundaries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Basic Stream Parsing and the /Length Entry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Building an Object Table for Indirect References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Handling Cross-References Between Objects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 5: Cross-Reference Tables, Streams, and the Trailer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The %%EOF Marker and Backward Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The Trailer Dictionary: Root, Size, Encrypt, Prev

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Traditional Cross-Reference Tables: Entry Formats and Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Parsing the xref Section Byte by Byte

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Cross-Reference Streams (PDF 1.5+): W, Index, Filter Entries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Incremental Updates and Multiple xref Sections

Chapter 6: Stream Filters and Decompression

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The Filter Chain: Sequential Decompression

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

ASCIIHexDecode and ASCII85Decode Implementations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

LZWDecode: TIFF Class F Algorithm Details

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

FlateDecode: zlib Integration and Raw Deflate

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

RunLengthDecode and CCITTFaxDecode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

DCTDecode, JBIG2Decode, JPXDecode Overview

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The Crypt Filter and Encryption Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Putting It All Together: Filter Chain Application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 7: Document Catalog and Page Tree Navigation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The Catalog Dictionary: Type, Pages, Names, Outlines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The Pages Object: Kids, Count, Parent Relationships

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Traversing the Page Tree Recursively

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Page Objects: MediaBox, CropBox, Rotate, Resources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Resource Dictionaries: Fonts, XObjects, ColorSpaces, Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Building a Document Navigation API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 8: Content Streams and Graphics Operators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Content Stream Syntax and Operator Tokens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The Current Transformation Matrix (CTM)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Path Construction Operators: moveto, lineto, curveto

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Path Painting: Stroke, Fill, Clip Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Text State and Positioning Operators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Graphics State Saving and Restoring: q and Q

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Marked Content and Tagged PDF Basics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Parsing Content Streams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Inline Images in Content Streams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 9: Coordinate Systems and Page Geometry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

User Space vs. Device Space

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The Transformation Matrix Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Clipping Paths and the Clip Operator Family

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Tiling Patterns and Pattern Paint

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Shadings: Axial, Radial, and Coons Mesh

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Page Boxes: MediaBox, CropBox, BleedBox, TrimBox, ArtBox

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 10: Text Rendering and Fonts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The Text Rendering Model: ShowText Operators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Font Dictionaries: Type, Subtype, Encoding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Standard Type 1 Fonts and Built-in Encodings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

TrueType and OpenType Font Embedding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

CID Fonts and CMap Parsing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Character Spacing, Word Spacing, Leading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Extracting Text from Content Streams

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 11: Images and Color Spaces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Image XObjects: Width, Height, BitsPerComponent

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Image Masking and Soft Masks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Device Color Spaces: RGB, CMYK, Gray

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

ICC-Based Color Spaces and Profile Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Special Color Spaces: Separation, DeviceN, Indexed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Loading and Decoding Image Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 12: Annotations, Links, and Interactivity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Annotation Dictionaries: Type, Subtype, Rect

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Link Annotations and URI Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Text and Highlight Annotations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Destination Types: XYZ, Fit, FitH, FitV, FitR, FitB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Outlines (Bookmarks): Tree Structure and Navigation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Named Destinations and the Names Dictionary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 13: Forms, AcroForms, and Interactive Fields

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The AcroForm Dictionary and Field Hierarchy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Field Types: Text, Button, Choice, Signature

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Field Appearance Streams and Rendering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Field States: ReadOnly, Required, Export Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Radio Button Groups and Checkboxes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Form Submission Actions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 14: Security, Encryption, and Digital Signatures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The Encrypt Dictionary and Security Handlers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Standard Security Handler: User/Owner Passwords

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

RC4 Encryption Algorithm Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

AES Encryption (AESV2, AESV3)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Integrating Encryption into the Core Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Permission Flags and Document Restrictions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Digital Signatures: Byte Range, Certificate Chains

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 15: Advanced Features in PDF 2.0

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

What Changed in PDF 2.0

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Enhanced Transparency and Blending

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Improved ICC Color Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

New Filter Capabilities and Requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Structural Improvements: Object Streams Refinements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Removed Features: XFA Deprecation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

UTF-8 Support and Text String Improvements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Portfolio and Collection Enhancements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Chapter 16: Robustness, Performance, and Real-World Issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Malformed Document Recovery Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Fuzz Testing and Security Hardening

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Memory Management: Caching Indirect Objects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Lazy Loading and On-Demand Decoding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Random Access Patterns and File I/O Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Interoperability Issues Across PDF Generators

Conclusion: A Complete Parser, An Open Format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

What We Built: The Complete Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Lessons from Building a Standards-Compliant Parser

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

Extending the Reader: Rendering and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

The Future of PDF in a Digital World

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/masteringpdfparsing.