Structured Analysis

sd04

Summary: In this module you will see the basics of structured analysis

Version: 4.2

Contents

Chapter I

Preamble

Hello, and welcome to your first Structured Development workshop!

This workshop is part of a mini-course for *42* students curated by *Prof. Enrico Vicario* and *Dr. Nicolò Pollini.* Throughout this course, you'll be introduced to a programming methodology that originated in the 1960s and has since evolved in tandem with the C language.

Structured Development is built on three key pillars:

- **Structured Programming**: This concerns the act of writing code in a clean, readable, and maintainable way. It emphasizes avoiding common pitfalls and code smells that lead to bugs and wasted time, especially during debugging.

- **Structured Design**: Good software design means organizing your codebase so that each function has a clear, well-defined purpose. The goal is to prevent situations where small changes ripple through the entire system, forcing you to rewrite large portions of code (yes, we're looking at you, *spaghetti code*).

- **Structured Analysis** *(today's focus)*: This is the most abstract level. It involves designing the architecture of an entire software system, especially when highly complex, based on its requirements and the flow of information needed to fulfill them.

> **Remember:** Code is not just written to work, it's written to be used. If it's worth using, it's worth writing for someone else to read, understand, and maintain. You never write code *just* for yourself.

Chapter II

Introduction

Structured Analysis is a systematic methodology used to understand how to build complex software systems by modeling how data flows and transforms across its various processes. It emphasizes creating artifacts such as:

- *Data Flow Diagrams* to show sources, sinks, and transformations of information;

- *Data Dictionaries* to define the structure and constraints of every data element;

- *Process Specifications* to describe each function's inputs, outputs, and internal logic.

By decomposing the system into well-defined processes and data stores before coding, Structured Analysis helps ensure that requirements are complete, consistent, and traceable, laying a solid foundation for reliable, maintainable design and implementation.

In the **June 20** lecture, you'll receive more in-depth information about Structured Analysis.
approach the exercises with an open mind and see how it goes. At the end of the course, you-ll receive feedback on *both* versions. You're encouraged to explore freely: discuss ideas with your classmates, search for insights online, and feel free to use AI tools if they help you reason better.

However, a word of caution about **Large Language Models (LLMs)** like ChatGPT, Gemini, and similar tools:
Researchers (including us) have studied the impact of over-relying on AI in learning-driven development, and the findings suggest that, while LLMs can boost your short-term performance, **overusing them can hinder your long-term growth**. The knowledge and skills you build now are what your future success depends on.
Here's our recommended approach:

- **Start by solving the problem on your own.**

- **If you get stuck, ask your peers or look for help online.**

- **If you're still stuck, and no one can help, use LLMs to understand *how* to approach the problem**, but don't let them solve it for you, unless you're certain that solving this particular problem is not important to your learning journey.

Chapter III

General instructions

- This document contains **7 C programming exercises**, arranged in **increasing order of difficulty**. While you're free to tackle them in any order, we recommend progressing sequentially to get the most out of the experience.

- These exercises are **not meant to be completed in full by everyone**, so don't be discouraged if you struggle early on. This course brings together students with a wide range of experience, and the more advanced exercises are designed to challenge even seasoned programmers.

- **Tomorrow, we'll release the fourth set of exercises (Rollback - Structured Development)**. So focus on doing your best with today's set, and try to apply what you learned during the lecture.

Chapter IV

Evaluation criteria

Unlike in previous experiences, you will be evaluated not only on whether your solution works and meets the requirements, but also on the **reasoning behind your choices**. In fact, this is the primary focus of the evaluation. *(No pressure, though)*
Here are a few important notes:

- ~~Your code must compile.~~ (DOES NOT APPLY)

- **You're not required to follow the 42 Norm**, but it's a good starting point if you're unsure about what constitutes clean and readable code.

> ⚠ **Note:** This is the **first version** of this document, so it may contain errors or inaccuracies. If you spot any issues, please let the staff know as soon as possible. Thank you!

Chapter V

Exercise 00 : Community Digital Library

	Exercise 00
	Exercise 00 : Community Digital Library
Turn-in directory : *ex*00/	
Files to turn in : `*.h, *.png, *.jpg, *.mdj, *.md, *.txt` [you may use other extensions as you see fit]	
Allowed functions : `free, malloc, get_next_line, ft_printf` or any equivalent you coded, all the functions present in your Libft	

You are tasked with creating a small console-based software for a community library. As of today, the staff of the library maintain a simple plain-text catalog of books in a file: each line contains information about one book, with fields separated by commas, for example an identifier number, then the book's title, then the author's name. Over time this file may grow to contain up to around a thousand entries. Occasionally, some lines might be malformed or contain extra whitespace. The staff want a program that, when run, reads this catalog file, handles any unexpected or malformed lines (e.g., by warning or skipping them), and keeps all valid entries in memory.

Once the catalog is loaded, the program should interactively prompt the user to choose how to look up books: either by searching for a substring in the title or by searching for a substring in the author's name. The input for the search should automatically trim any trailing spaces or newlines, and the search must be case-insensitive, so that typing "kernighan" will correctly match "Kernighan". If no matching books are found, the program should inform the user; otherwise it should display each matching entry's identifier, title, and author in a clear format. After showing results, the program should clean up any allocated resources and exit.

The tool must be invoked from the command line with the catalog filename as its argument. If the file cannot be opened, the program should report an error and terminate. Internally, the program will keep up no more than 1000 book records in memory; The user interface is text-based: after loading, prompt clearly for "search by title" or "search by

author" requiring a valid input (e.g., only accept "1" or "2" or equivalent), then prompt for the substring, re-prompting if the user just presses Enter without typing anything meaningful, and take into account a way for the user to terminate the program correctly.

> This text is difficult to interpret, isn't it?

You don't need to implement this project, but this is a great opportunity to experience what it's like to interpret the customer's language. In 42 Subjects, the challenge lies in deciphering dense but unambiguous specifications: every detail is there, once you learn the *jargon*.
Here, the situation is reversed: **you're the one who knows the jargon**, and the client is speaking plain, informal English. **It's your responsibility to translate their ideas into a structured technical plan.**

Important:
For this exercise, your goal is to design the high-level structure of the project. The expected output is **a set of header files** containing the key data structures and function prototypes you would use if you were to implement the full project.

Pro tip: he original problem description alone probably won't be enough.
Take some time to extract and rephrase the core requirements (e.g., in a bullet list style) and sketch out your interpretation of the system using schemes, flowcharts, use-cases, and any other tools you find useful.
You don't need to use formal tools: **pictures of plain paper are perfectly fine**, as long as they're readable.

You're welcome to submit these artifacts as well, just be sure to include a short explanation of what they are and why they help clarify your design.

Chapter VI

Exercise 01: Expense Tracker

	Exercise 01
	Exercise 01: Expense Tracker
Turn-in directory : *ex*01/	
Files to turn in : `*.h, *.png, *.jpg, *.mdj, *.md, *.txt [you may use other extensions as you see fit]`	
Allowed functions : `free, malloc, get_next_line, ft_printf or any equivalent you coded, all the functions present in your Libft`	

You have been approached by a young professional who keeps track of their spending in a simple text file and wants a small console program to help them make sense of it. Each line of their file represents one purchase or expense, written by hand or exported from another app; they separate fields with commas but sometimes forget to format things perfectly. A typical line might look like '2024-01-15,12.50,Food,Latte at cafe', but occasionally there are typos, extra spaces, missing fields, non-numeric amounts, or dates in the wrong format.

They ask you to build a software to read their file (whose name is supplied on the command line) and quietly skip invalid lines (or printing a warning so they know something was off) while storing up to around a thousand valid records in memory. After loading, the program should ask the user how they want to view or filter their expenses: perhaps by entering a piece of a category name (like "food" to catch "Food" or "food"), or by specifying a date range (e.g., between "2024-01-01" and "2024-01-31"), or simply choosing to see everything. The prompts must insist on valid input: re-asking if someone just presses Enter, or types a date in a wrong format, or gives a start date that comes after the end date.

Internally, dates are stored in the 'YYYY-MM-DD' format, allowing for lexicographic comparison once validated. Amounts are positive floating-point numbers. If a line contains 'invalid_amount', 'zero', or a negative value, that record should be skipped with a warning. Categories and descriptions may contain extra whitespace or mixed casing; your program should trim them and handle category comparisons in a **case-insensitive** manner. In any summary, categories that differ only in case should be grouped together as identical. After the user selects a filter, the program filters the in-memory records

accordingly, then computes and displays a summary:

- Total number of records under consideration

- Total amount spent

- Average expense

- Breakdown by category, showing:
    - Total spent per category
    - Percentage of the total each category represents

If no records match the filter, the program should clearly indicate that. Once the summary is displayed, the program should clean up any allocated memory and exit. If the file cannot be opened, the program should print an error message and terminate.

Important:

For this exercise, your goal is to design the high-level structure of the project. The expected output is **a set of header files** containing the key data structures and function prototypes you would use if you were to implement the full project.

Pro tip: he original problem description alone probably won't be enough.
Take some time to extract and rephrase the core requirements (e.g., in a bullet list style) and sketch out your interpretation of the system using schemes, flowcharts, use-cases, and any other tools you find useful.
You don't need to use formal tools: **pictures of plain paper are perfectly fine**, as long as they're readable.

You're welcome to submit these artifacts as well, just be sure to include a short explanation of what they are and why they help clarify your design.

Chapter VII

Exercise 02: Contact Manager

	Exercise 02
	Exercise 02: Contact Manager
Turn-in directory : *ex02/*	
Files to turn in : `*.h, *.png, *.jpg, *.mdj, *.md, *.txt` [you may use other extensions as you see fit]	
Allowed functions : `free, malloc, get_next_line, ft_printf or any equivalent you coded, all the functions present in your Libft`	

You've been asked to build a simple console-based contact management tool for a small organization. Contacts are stored in a plain-text CSV file whose name is given as the single command-line argument when starting the program. Each line should hold six comma-separated fields: an integer ID, a name, phone, email, city, and a free-form address. In practice the file may contain up to around a thousand valid contacts but also malformed lines: missing fields, extra whitespace, invalid formats, or duplicate IDs, so your program must open the file (or error and exit if it can't), read it line by line, trim whitespace around fields, and apply basic validation:

- the ID must parse to a positive integer not already loaded;

- name and city must be non-empty after trimming;

- phone should use only digits and typical symbols and include at least one digit;

- email must have one '@' with non-empty local and domain parts and at least one dot in the domain;

- address may be empty;

- invalid lines are skipped with a warning but don't crash the program;

- valid contacts are stored in memory up to capacity, tracking the maximum ID.

After loading, enter a menu loop offering options like listing all contacts, searching by name or city substring, adding a new contact, updating an existing contact, deleting a contact, and saving & exiting.

On save, write all contacts back to the same filename (tip: use a temporary file and then rename, reporting any write errors) and exit. Every prompt must re-ask on invalid input; after each operation return to the menu until save-and-exit is chosen.

Important:

For this exercise, your goal is to design the high-level structure of the project. The expected output is **a set of header files** containing the key data structures and function prototypes you would use if you were to implement the full project.

Pro tip: he original problem description alone probably won't be enough.
Take some time to extract and rephrase the core requirements (e.g., in a bullet list style) and sketch out your interpretation of the system using schemes, flowcharts, use-cases, and any other tools you find useful.
You don't need to use formal tools: **pictures of plain paper are perfectly fine**, as long as they're readable.

You're welcome to submit these artifacts as well, just be sure to include a short explanation of what they are and why they help clarify your design.

Chapter VIII

Exercise 03: Movie Watchlist

	Exercise 03
	Exercise 03: Movie Watchlist
Turn-in directory : *ex03/*	
Files to turn in : `*.h, *.png, *.jpg, *.mdj, *.md, *.txt` [you may use other extensions as you see fit]	
Allowed functions : `free, malloc, get_next_line, ft_printf` or any equivalent you coded, all the functions present in your Libft	

An emergent streaming platform hires you to create a console-based Movie Watchlist app that reads from and writes to a plain-text CSV file given by the user on the command line. Each line in that file is expected to hold six comma-separated fields: a positive integer ID, a movie title, a genre, a watched flag (0 or 1), a rating (1-10 if watched, otherwise 0), and a date watched in YYYY-MM-DD format if watched (otherwise empty). In practice, the file may contain up to around a thousand valid records but also occasional malformed lines (e.g., missing fields, extra whitespace, invalid IDs or duplicates, incorrect flags, out-of-range ratings, or bad dates), so your program should attempt to open the file, read it line by line, trim whitespace around each field, and apply basic validations:

- the ID must parse to a positive integer not seen before;

- title and genre must be non-empty;

- watched flag must be exactly 0 or 1;

- if marked watched, rating can be an integer from 1 to 10 and date watched must match YYYY-MM-DD within plausible ranges;

- if unwatched, any rating or date content is ignored or warned about but not fatal;

- invalid lines are skipped with a warning to stderr but do not crash the program;

- valid entries are stored in memory, up to capacity, tracking the maximum ID encountered so new entries can be assigned unique IDs.

After loading, the program enters an interactive menu loop offering operations such as listing all movies in ascending ID order, searching by title or genre substring, filtering by watched or unwatched status, adding a new movie, marking an existing movie as watched, updating title or genre of a chosen movie, deleting a movie (prompting for confirmation), showing summary statistics (e.g., total movies, counts watched vs. unwatched, average rating of watched and breakdowns by genre for watched and unwatched sets), and finally saving and exiting. Every prompt must validate the inputs and re-prompt if invalid.

When saving, all records currently in memory are written back to the same filename, so that the watchlist on disk reflects any additions, updates, or deletions.

Important:

For this exercise, your goal is to design the high-level structure of the project. The expected output is **a set of header files** containing the key data structures and function prototypes you would use if you were to implement the full project.

Pro tip: he original problem description alone probably won't be enough.
Take some time to extract and rephrase the core requirements (e.g., in a bullet list style) and sketch out your interpretation of the system using schemes, flowcharts, use-cases, and any other tools you find useful.
You don't need to use formal tools: **pictures of plain paper are perfectly fine**, as long as they're readable.

You're welcome to submit these artifacts as well, just be sure to include a short explanation of what they are and why they help clarify your design.

Chapter IX

Exercise 04: Expense Tracker (bonus)

	Exercise 04
	Exercise 04: Expense Tracker (bonus)
Turn-in directory : *ex04/*	
Files to turn in : `*.h, *.png, *.jpg, *.mdj, *.md, *.txt` [you may use other extensions as you see fit]	
Allowed functions : `free, malloc, get_next_line, ft_printf` or any equivalent you coded, all the functions present in your Libft	

In 'ex01' you were tasked with a read-only expense summarizer that loaded a file of date, amount, category, and description lines and showed filtered summaries. The client has come back with new requests:

"It would be great if I could not only view past expenses but also add new entries and have the program update the file accordingly. But what if I enter something wrong or want to delete an expense later?"

Since **"It would be great if I could ..."** means **"I want to ... and you should build it"**, your software must now add entries on the fly and also correct or remove mistakes after loading. Your enhanced Expense Tracker must still begin by opening the given file and reading each line, but now it should detect whether a line has four fields (old format) or five fields (new format, including an ID). For old-format lines you assign sequential IDs starting at 1 (or continuing from the highest existing ID), while for new-format lines you parse and preserve the positive integer ID if it does not collide with others. Invalid lines (e.g., wrong field count, bad date format, non-positive or non-numeric amounts, empty category, or malformed ID) are skipped with a warning. Valid records are stored in memory up to a capacity (around 1000), tracking the maximum ID seen so that newly added expenses get unique IDs beyond that.

Once loaded, the program enters an interactive loop offering operations in a menu:

- listing all expenses in ascending ID order with their date, amount, category, and description;

- searching or filtering by category substring or a date range;

- adding a new expense by prompting for date, amount, category, and description (optional);

- editing an existing expense, showing current values and letting the user update it or not;

- deleting an expense by ID with confirmation, removing it from memory;

- showing a summary over either all or a filtered subset of records, computing total count, total spent, average expense, and a breakdown by category with totals and percentages;

- every user prompt must validate input and re-prompt if invalid.

When the user chooses to save and exit, the program writes all in-memory records back to the same filename, outputting each line in the new five-field format, so that future runs can load and preserve edits and deletions.

Important:

For this exercise, your goal is to design the high-level structure of the project. The expected output is **a set of header files** containing the key data structures and function prototypes you would use if you were to implement the full project.

Pro tip: he original problem description alone probably won't be enough.
Take some time to extract and rephrase the core requirements (e.g., in a bullet list style) and sketch out your interpretation of the system using schemes, flowcharts, use-cases, and any other tools you find useful.
You don't need to use formal tools: **pictures of plain paper are perfectly fine**, as long as they're readable.

You're welcome to submit these artifacts as well, just be sure to include a short explanation of what they are and why they help clarify your design.

Chapter X

Exercise 05: Contact Manager (bonus)

	Exercise 05
	Exercise 05: Contact Manager (bonus)
Turn-in directory : *ex05/*	
Files to turn in : *.h, *.png, *.jpg, *.mdj, *.md, *.txt [you may use other extensions as you see fit]	
Allowed functions : free, malloc, get_next_line, ft_printf or any equivalent you coded, all the functions present in your Libft	

In 'ex02' you were tasked with a contact management tool for a small organization. The client returns mid-development with some requests to make the application more flexible for their growing needs.

"In addition to basic CRUD operations, we realized we"d need to categorize contacts into user-defined groups (for example "Team", "Vendors", or "Friends") so we can view or act on these groups more easily. Oh, and can you also add a simple birthday-tracking feature? Like an optional birthdate for each contact and, upon request, show a list of upcoming birthdays within the next month? That would be nice!"

These additions should fit naturally into the existing console-based tool without breaking backward compatibility: contacts stored previously (with the original six fields) should continue to load, but the data model and file format must evolve so that new or updated contacts can include group or birthday information.

When loading the CSV file, the program will need to detect whether a line includes the extra fields for groups or birthdate, or if it"s in the original format; for older entries that lack these fields, the application would assign defaults (e.g., no group, no birthdate). Validating and parsing a date-of-birth and groups requires the same care as existing fields: ensure it matches a logical pattern and/or is a plausible value. The UI menu should gain options like "List contacts by group", "Assign or change groups for a contact", and "Show upcoming birthdays", all with robust prompts and validation.

Saving must write out the extended fields so that on the next run the groups and birth-dates persist, but still handle older files by appending defaults when missing.

Important:

For this exercise, your goal is to design the high-level structure of the project. The expected output is **a set of header files** containing the key data structures and function prototypes you would use if you were to implement the full project.

Pro tip: he original problem description alone probably won't be enough.
Take some time to extract and rephrase the core requirements (e.g., in a bullet list style) and sketch out your interpretation of the system using schemes, flowcharts, use-cases, and any other tools you find useful.
You don't need to use formal tools: **pictures of plain paper are perfectly fine**, as long as they're readable.

You're welcome to submit these artifacts as well, just be sure to include a short explanation of what they are and why they help clarify your design.

Chapter XI

Exercise 06: Movie Watchlist (bonus)

	Exercise 06
	Exercise 06: Movie Watchlist (bonus)
Turn-in directory : *ex06/*	
Files to turn in : *.h, *.png, *.jpg, *.mdj, *.md, *.txt [you may use other extensions as you see fit]	
Allowed functions : free, malloc, get_next_line, ft_printf or any equivalent you coded, all the functions present in your Libft	

In 'ex03', you were tasked with building a Movie Watchlist app that tracked whether each movie was watched, alongside its rating and the date it was watched. But now the client is back with bigger ambitions:

"The Movie Market is rapidly expanding and so do we. We are not just a Movie streaming platform anymore, we now also provide TV Shows and even Anime, so our app must grow accordingly, can you make it happen? Oh, and one more thing: people often rewatch their favorite titles. Our data shows they'd love to keep track of every rewatch over time. Can the app handle that too?"

In other words, they want to keep a full viewing history instead of just marking a movie as watched once, and they also wish to track TV shows and anime alongside movies. The new tool must still load the existing watchlist file so that older entries continue to work, but it must interpret six-field movie-only lines as single-entry histories and recognize a new extended format that begins with a media-type label ("Movie", "TV", or "Anime") followed by ID, title, genre, and a serialized history of viewings. Any malformed or duplicate entries should be skipped with a warning, yet valid records, up to around a thousand items, are stored in memory with unique IDs preserved or assigned sequentially.

When the program starts, it reads each line, trims whitespace, detects the format type, and parses old-format lines into a movie item with at most one history entry or new-format lines into the appropriate media type with multiple viewings. Movie histories

consist of dates like "YYYY-MM-DD", while TV shows and anime histories consist of season-episode identifiers plus the date such as "2-5:2021-11-12". Usual validation is applied:

- dates must match the YYYY-MM-DD pattern;

- ratings be integers from 1 to 10;

- season and episode numbers must be positive integers;

- invalid IDs, empty titles or genres, malformed dates, out-of-range ratings, or bad history syntax cause that line to be skipped (with a warning) but do not abort loading;

- valid items are kept in an in-memory sorted list.

Once loaded, the program enters a console-based interactive loop offering separate menus for Movies, TV Shows, and Anime, each allowing the user to list all items of that type, add a new title with empty history, record a new viewing (prompting appropriately for date and rating, and for TV/Anime also season and episode), view the full history for a chosen item, edit its title or genre, delete an item with confirmation, and show summary statistics such as total items tracked, count with at least one viewing, total viewings or episodes watched, average ratings, and breakdowns by genre. There must also be a global search feature that asks for a substring and returns matches across all media types by title or genre. Every prompt enforces valid input and reprompts if invalid.

When the user chooses to save and exit, the program writes all items back to the same filename, writing each line in the extended format or left blank if empty.

Important:

For this exercise, your goal is to design the high-level structure of the project. The expected output is **a set of header files** containing the key data structures and function prototypes you would use if you were to implement the full project.

Pro tip: he original problem description alone probably won't be enough.
Take some time to extract and rephrase the core requirements (e.g., in a bullet list style) and sketch out your interpretation of the system using schemes, flowcharts, use-cases, and any other tools you find useful.
You don't need to use formal tools: **pictures of plain paper are perfectly fine**, as long as they're readable.

You're welcome to submit these artifacts as well, just be sure to include a short explanation of what they are and why they help clarify your design.

Chapter XII

Submission and peer evaluation

Create a personal repository named:
    42xunifi-structured-development-2025-<your-intra-login>
Share it with the staff and organize the exercises as follows:

```
sd00/
 ex00/
 ex01/
 ex02/
 ...
sd01/
 ex00/
 ex01/
 ...
sd02/
 ex00/
 ...
```

You can share your repository by filling out this form (Only if you haven't done so already)

- **Update your repository regularly**, and make sure to **push all completed exercises before the deadline**.

- **This workshop's deadline is June 20**. Any changes made after this date **will not be considered** for assessment purposes.

- This exercise follows the **42 methodology**, so **peer-to-peer collaboration is allowed and strongly encouraged**.

- **Formal peer evaluations via Intra will not be available**, but you are still welcome to ask your peers for feedback on your work.

Our feedback won't be immediate, and unfortunately we've reached a level of abstraction so high that it's difficult to even think of an automated tool capable of providing meaningful feedback on something so layered and conceptual. Tools like **SonarQube** and **CCCC** still apply, but they operate at a lower level of abstraction. To help you track your progress, we recommend using AI tools (i.e., any LLM of your choice) to get feedback on your design ideas. For example:

- *"In the context of Structured Analysis (as formalized by Tom De Marco in 1978), I was given the following problem statement.*

- *I was thinking of structuring the project like this… and then that…*

- *What do you think?*

- *Here's the problem statement I was given: <insert-problem-statement-here>."*

We can't stress this enough:

- **Don't let AI dictate your actions**, always start by thinking through a solution on your own, then use AI for feedback and refinement.

- **Keep things simple**, you're here to *learn* how to engineer software, no one expects you to *be* a software engineer after just one week.

- **Don't fall down the rabbit hole**, if something gets too complex, move on to the next task. Aim to produce at least *something* for each exercise.

Chapter XIII

Setting Up SonarQube (SonarCloud) with GitHub

1. **Create a public GitHub repository** for the course, named:
   `42xunifi-structured-development-2025-<your-intra-login>`.

2. Go to [SonarCloud](#) and click **"Start now."**

3. Sign up using your **GitHub account**.

4. When prompted, **import an organization**. You can use `"42"` or your GitHub username.

5. **Authorize access only to selected repositories**, and choose the one you created in step 1.

6. Choose a **Name** and a **Key** for your organization (your username is usually fine for both).

7. Select the **Free Plan** when prompted.

8. Click **"Create Organization."**

9. Select the previously created repository and click **"Set Up."**

10. For code technology, choose **"Previous version."**

11. Click **"Create Project."**

12. *Profit.*