Extracted from:

Resourceful Code Reuse

Write → Compile → Link → Run

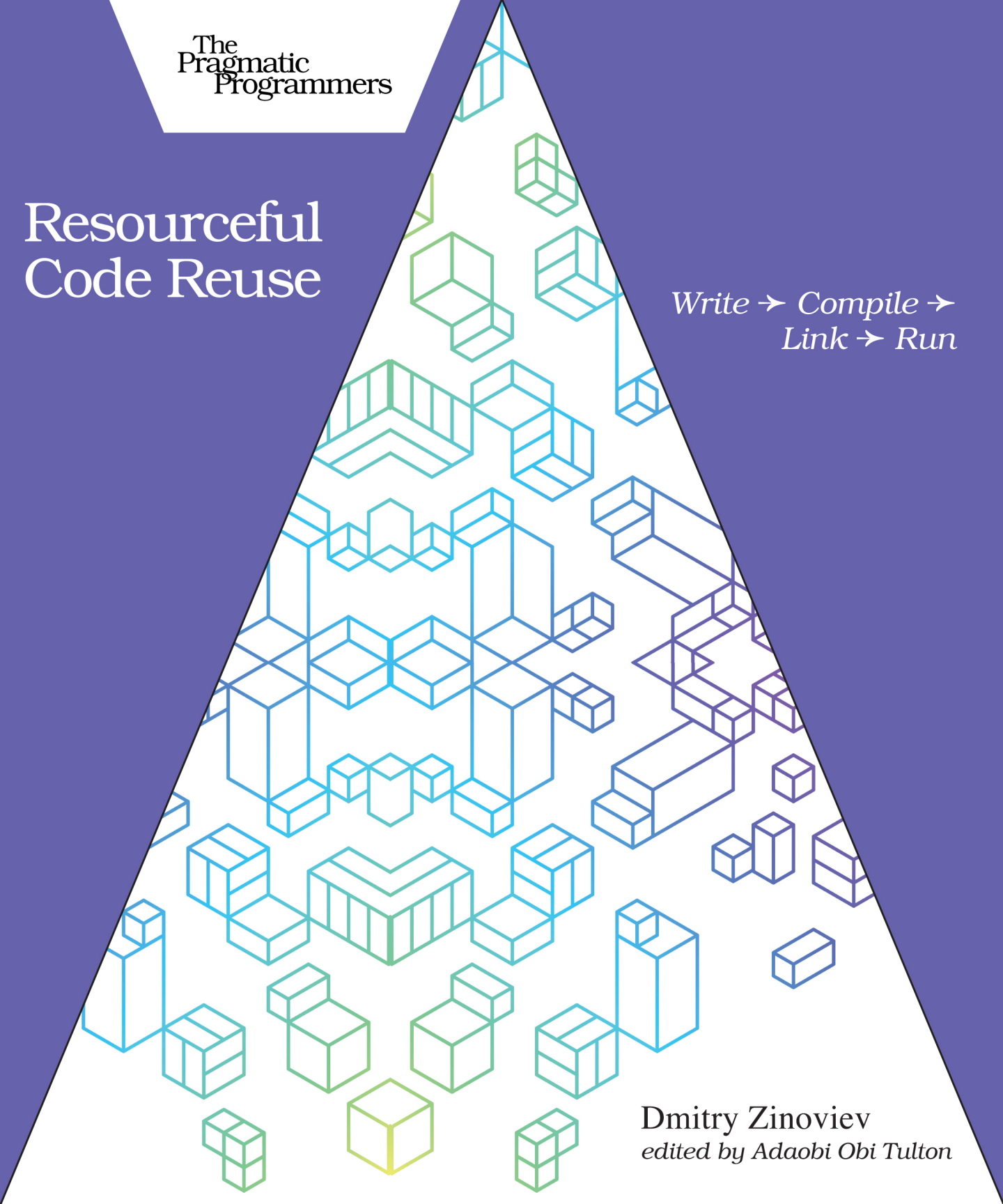
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Resourceful Code Reuse

Write → Compile → Link → Run



Dmitry Zinoviev
edited by Adaobi Obi Tulton

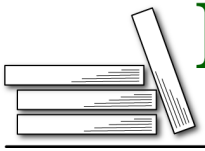
Resourceful Code Reuse

Write → Compile → Link → Run

Dmitry Zinoviev

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Adaobi Obi Tulton
Copy Editor: Corina Lebegioara
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-820-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—April 2021

Arranging Source and Header Files (the C Way)

Proper code reuse can be challenging because it requires a specific mind-set. A common mistake that even professional programmers make is putting all their code in the same file like this:

jsontool.c

```
Line 1  #include <stdio.h>   // For input/output
        #include <stdlib.h>  // For EXIT_SUCCESS or EXIT_FAILURE
        #include <stdbool.h> // For bool, false, and true
        
      5 typedef struct JSON_Object {
          // Lots of lines of code here
          // ...
        } JSON_Object;
        
     10 // Lots of helper functions, data types, and global variables
        int json_errno = 0;
        bool read_boolean(FILE* infile) { /* ... */; return false; }
        char *read_string(FILE* infile) { /* ... */; return NULL; }
        void *obscure_helper() { /* ... */ ; return NULL; }
     15 // ...
        
        // JSON parser and writer
        JSON_Object *read_json(char *fname) {
          JSON_Object *object = NULL;
     20   // Lots of lines of code here
          // ...
          return object;
        }
        
     25 int write_json(char *fname, JSON_Object *object) {
          int status = 0;
          // Lots of lines of code here
          // ...
          return status;
     30 }
        
        // Business logic
        static JSON_Object *do_stuff(JSON_Object *json) {
          // Lots of lines of code here
     35   // ...
          return json;
        }
        
        int main() {
     40   JSON_Object *json = read_json("their_file.json");
          if(!json) return EXIT_FAILURE;
        
          // Do some processing
          json = do_stuff(json);
```

```
45
-   if(!write_json("my_json.json", json))
-     return EXIT_FAILURE;
-
-   puts("Success!");
50  return EXIT_SUCCESS;
-  }
```

These programmers will then compile the program with their favorite C compiler, get an executable file, and run it:

```
/home/dzreuse/> cc jsontool.c -o jsontool
/home/dzreuse/> ./jsontool
Success!
```

Code reuse aside, this is an excellent C program! It's (hopefully) correct, concise, and easy to write, and it follows natural human logic. The program defines a data structure, implements helper functions, uses them to implement the parser and the writer, and uses the parser and the writer to solve the problem.

Sadly, it has two issues: it's inefficient from a C compiler's point of view, and it's virtually not reusable.

A C compiler always translates one compilation unit at a time, typically the whole C file with all header files (such as stdio.h) pasted into it verbatim. If you change only one line or character in a C or header file, the whole unit must be recompiled. As your program grows, compilation takes longer. If you want to make your program compile faster, you must make compilation units smaller. I'll show you how to take advantage of this splitting in by using what is known as *separate compilation* in Compiling Object Files, on page ?.

But let's get to the point of reusability. Why is the code on page 1 not suitable for reuse? Because it solves a specific problem: it parses a JSON file, transforms it in a specific way, and writes it into another file. Since the code is an indivisible compilation unit, no function from the file can be used in any other project except by copying and pasting it into a text editor—but copying and pasting code hurts its maintainability and consistency and blunts Occam's razor. And since the code already has the main() function, it cannot become a part of any other project as a whole.

Here's the bottom line: a monolithic program file written in the C language is slow to compile and hard (frankly, impossible) to reuse. let's break the compilation unit into several smaller units and see if it helps.

A long and tedious list on page ? goes to great lengths to enumerate the ways of decomposing *components* (functions, data types, and global variables)

into somewhat coherent collections. For now, if your file isn't insanely large yet, you can split it into a more reusable input/output part (JSON support) and a non-reusable business logic part (JSON processing). Use your best judgment to estimate each function's propensity to be included in another project in the future. Ask yourself: "How likely is it that I or someone else I know or can think of will require the operation implemented in this function?" If the answer is "more than likely," that function belongs to the reusable part. If the answer is "unlikely," it belongs to the non-reusable part. It's okay to make a mistake! Splitting a compilation unit is a *wicked problem*: it doesn't have one correct answer.

Oh, and place *all* functions called from any potentially reusable function into the same reusable part or provide a way to combine parts. A function won't work without its dependencies!

On a closer look, the non-reusable part probably consists of the `main()` function and the function that implements the business logic of the `jsontool`. Let's call that part "business logic." The helper functions, the parser, and the writer are likely in the reusable part. You may further subdivide the reusable part into two more parts.

A programmer may use the helper functions that read and write strings, numbers, Boolean values, and the like in any project that requires reading such elements—this is our "JSON tokens" part. And just as "J" in "JSON" doesn't mean JavaScript anymore, "JSON" in "JSON tokens" doesn't refer exclusively to JSON tokens but to any JSON-like tokens.

In contrast, the parser and the writer are JSON-specific. Their applicability area is JSON processing; they're useless in a non-JSON project and belong to a "JSON syntax" part.

Let's not forget that the "JSON syntax" part depends on the "JSON tokens" part. The latter may exist on its own but not the other way around. If you plan to reuse the "JSON syntax" components, you should reuse the "JSON tokens" too.

You'll have to apply the same classification procedure to the data types. Once the components are nicely categorized, we'll use a few technical tricks to separate the compilation unit into three units by following these steps:

1. Physically move the content into three files.
2. Hide the global variables that aren't global anymore.
3. Create a header file: an interface to the new source files.