

Session 16

Find structured-programming violations in a small program and rewrite it with sequence, selection and iteration only

This session takes a small program written earlier (a menu-driven bank balance in C), checks it against the structured programming paradigm, lists every construct that breaks the paradigm with a line reference, and rewrites it so that only sequence, selection and iteration remain. The rewritten program produces byte-for-byte the same output on the same input, which is how you prove the rewrite changed structure and not behaviour. Structured programming is the reason code from Session 13 can be reviewed, tested and changed at all.

Objectives

✗ Do not copy. Read for understanding and the viva

- State the three control structures of structured programming and the single-entry, single-exit rule.
- Recognise `goto`, jumps out of nested loops, mid-function `return`, and flag variables as violations, and say which rule each one breaks.
- Rewrite a working program using `while`, `switch` and `if` only, one entry and one exit per function.
- Prove behaviour is unchanged by running both versions on the same input.

Problem Statement

■ Write in lab record

Session 16: Select a small portion of any program written by you. Check if the portion of code selected by you is having constructs that violate the structured programming paradigm. If yes, then rewrite the code to conform to structured programming paradigm. If no, check another portion of code.

Concept

✗ Do not copy. Read for understanding and the viva

The three structures

Bohm and Jacopini showed that any computable program can be written with only three control structures, each with one entry point and one exit point:

- Sequence: statements executed one after another.
- Selection: `if`, `if-else`, `switch`; choose one of several blocks, then continue after the block.
- Iteration: `while`, `for`, `do-while`; repeat a block while a condition holds, then continue after the loop.

A block built from these can be read top to bottom, and the state at any line depends only on the lines above it in that block. That property is what a code reviewer relies on.

What breaks the paradigm

Construct	Why it violates the rule
<code>goto</code> and labels	Control enters a block from anywhere; the block has many entries
<code>goto</code> out of nested loops	The loop has a second exit; the code after the loop cannot assume the loop condition became false
<code>return</code> in the middle of a function	The function has many exits; cleanup and postconditions are scattered
Flag variable set in one block and tested in another	Turns a selection into a hidden jump; the reader must trace every assignment to know what the test means
Falling through <code>if</code> chains where each branch jumps away	Selection without a join point; there is no line where "after the choice" begins

`break` inside a `switch` and a loop-controlling boolean in the `while` condition are accepted in practice because they keep one exit per structure. The pure form avoids even those; this session uses them because C has no other way to leave a `switch`.

How to rewrite

1. Every `goto` loop becomes a `while` with a condition variable.
2. Every `goto` out of a nested loop becomes a loop condition that includes "not yet found".
3. Every early `return` becomes an assignment to a status variable inside an `if-else` chain, with one `return` at the end.
4. Every flag tested far from where it was set is either deleted (the test moves next to the condition) or becomes the loop condition.
5. Repeated blocks under different labels become functions with one job each.

Code Selected for Review

■ Write in lab record

The portion is the complete `unstructured.c` (86 lines): a bank balance program with a menu for deposit, withdraw, show balance and exit. Withdrawals are paid out in 500 and 200 rupee notes, so an amount that cannot be formed from those notes is refused. The program works; it was written quickly and never reviewed.

unstructured.c

```
1  /*
2   * unstructured.c
3   * Menu-driven bank balance: deposit, withdraw, show balance, exit.
4   * Withdrawals are paid out in 500 and 200 rupee notes only.
5   * Works, but the control flow is built from goto, flags and early returns.
6   */
7  #include <stdio.h>
8
9  int balance = 0;
10 int flag = 0;      /* 0 = nothing, 1 = notes found, 2 = error already printed */
11
12 /* Returns 0 on success, -1 bad amount, -2 insufficient funds, -3 no notes *
13 int withdraw(int amount)
14 {
15     int fives, twos;
16     if (amount ≤ 0)
17         return -1;
18     if (amount > balance)
19         return -2;
20     flag = 0;
21     for (fives = amount / 500; fives ≥ 0; fives--) {
22         for (twos = 0; twos * 200 ≤ amount; twos++) {
23             if (fives * 500 + twos * 200 == amount) {
24                 flag = 1;
25                 goto found;
26             }
27         }
28     }
29 found:
30     if (flag ≠ 1)
31         return -3;
32     balance -= amount;
33     printf("Paid %d as %d x 500 and %d x 200\n", amount, fives, twos);
34     return 0;
35 }
36
37 int main(void)
38 {
39     int choice, amount, result;
40 menu:
41     printf("\n1 Deposit  2 Withdraw  3 Balance  4 Exit\nChoice: ");
42     if (scanf("%d", &choice) ≠ 1)
43         goto quit;
```

```
44     if (choice == 1)
45         goto deposit;
46     if (choice == 2)
47         goto withdraw;
48     if (choice == 3)
49         goto show;
50     if (choice == 4)
51         goto quit;
52     printf("Invalid choice\n");
53     goto menu;
54 deposit:
55     printf("Amount: ");
56     if (scanf("%d", &amount) != 1)
57         goto quit;
58     flag = 0;
59     if (amount <= 0) {
60         printf("Invalid amount\n");
61         flag = 2;
62     }
63     if (flag == 2)
64         goto menu;
65     balance += amount;
66     printf("Deposited %d\n", amount);
67     goto menu;
68 withdraw:
69     printf("Amount: ");
70     if (scanf("%d", &amount) != 1)
71         goto quit;
72     result = withdraw(amount);
73     if (result == -1)
74         printf("Invalid amount\n");
75     if (result == -2)
76         printf("Insufficient balance\n");
77     if (result == -3)
78         printf("Amount cannot be paid in 500 and 200 notes\n");
79     goto menu;
80 show:
81     printf("Balance: %d\n", balance);
82     goto menu;
83 quit:
84     printf("Bye\n");
85     return 0;
86 }
```

Violations Found

■ Write in lab record

Line(s)	Construct	Which rule it breaks	How it was rewritten
10	Global <code>flag</code> with three meanings (0 nothing, 1 notes found, 2 error printed), set in <code>withdraw()</code> and in <code>main</code>	Flag variable spaghetti: a selection made in one place is tested in another; the reader must trace every write to know what <code>flag = 2</code> means	Deleted. <code>split_notes()</code> uses a local <code>found</code> as its loop condition (structured.c line 23); the deposit check is a plain <code>if-else</code> in <code>deposit()</code> (line 56)
16 to 19	Two <code>return</code> statements before the main work of <code>withdraw()</code>	Multiple exits from a function; the postcondition “balance was reduced” is true on one path and false on two others with no single place to see that	<code>withdraw()</code> (line 36) sets <code>status</code> in an <code>if-else</code> chain and has one <code>return status</code> on line 53
21 to 28	Nested <code>for</code> loops searching note combinations, left with <code>goto found</code> on line 25	Iteration with two exits: the loop ends either when the condition fails or when the jump fires, and lines 30 to 33 use <code>fives</code> and <code>twos</code> whose values depend on which exit was taken	<code>split_notes()</code> (line 18) has one <code>while</code> whose condition is <code>!found && fives ≥ 0</code> ; the inner loop is replaced by an arithmetic test (<code>rest % 200 == 0</code>), so there is nothing to jump out of
29	Label <code>found:</code> inside a function body	A second entry point into the block after the loop	Removed with the <code>goto</code>
30 to 31	Third <code>return</code> in the middle of <code>withdraw()</code>	Same as lines 16 to 19	Same <code>status</code> variable, single exit

Line(s)	Construct	Which rule it breaks	How it was rewritten
40, 54, 68, 80, 83	Labels <code>menu</code> , <code>deposit</code> , <code>withdraw</code> , <code>show</code> , <code>quit</code> in <code>main</code>	The whole of <code>main</code> is a set of blocks with multiple entries; there is no loop, so the reader cannot see that the menu repeats	<code>main</code> (line 83) is one <code>while</code> (running) loop (line 89) containing one <code>switch</code> (choice) (line 94); the menu repeats until <code>running</code> becomes 0
44 to 51	Chain of <code>if</code> (<code>choice = n</code>) <code>goto</code> <code>label</code>	Selection without a join point: each branch jumps away, so there is no line where "after the choice" begins	<code>switch (choice)</code> with <code>case 1</code> to <code>case 4</code> and <code>default</code> ; every case ends with <code>break</code> and control joins at the bottom of the loop body
43, 51, 57, 71	<code>goto quit</code> from four places	Four exits from the main loop scattered through the body	<code>running = 0</code> at lines 92, 98, 106 and 115; the loop condition is the single exit and <code>printf("Bye")</code> runs exactly once after it
53, 64, 67, 79, 82	<code>goto menu</code> to restart the loop	Iteration built by hand with jumps; the loop has five back-edges instead of one	The <code>while</code> loop has one back-edge at its closing brace
58 to 64	<code>flag = 0; if (bad)</code> <code>flag = 2; if (flag</code> <code>= 2) goto menu;</code>	A flag used to delay a decision by three lines; the value 2 has no meaning outside this block	<code>deposit()</code> line 56: <code>if</code> (<code>amount ≤ 0</code>) <code>error</code> <code>else</code> <code>deposit</code> , no flag
73 to 78	Three independent <code>if (result = n)</code> tests on one variable	Selection written as three sequences; a reader cannot tell at a glance that the cases are exclusive	<code>report_withdraw()</code> line 66 uses one <code>switch (result)</code>

Every violation is one of four kinds: `goto`, jump out of a loop, mid-function `return`, or a flag carried across blocks. The rewrite has none of them.

Rewritten Program

■ Write in lab record

structured.c

```
1  /*
2   * structured.c
3   * Same bank balance program as unstructured.c, rewritten with only
4   * sequence, selection (if, switch) and iteration (while). Every function
5   * has one entry and one exit. No goto, no flags carried across blocks.
6   */
7  #include <stdio.h>
8
9  #define NOTE_BIG    500
10 #define NOTE_SMALL  200
11
12 static int balance = 0;
13
14 /*
15  * Finds how many 500 and 200 notes add up to amount.
16  * Returns 1 and fills *big, *small when possible, else returns 0.
17  */
18 static int split_notes(int amount, int *big, int *small)
19 {
20     int found = 0;
21     int fives = amount / NOTE_BIG;
22
23     while (!found && fives ≥ 0) {
24         int rest = amount - fives * NOTE_BIG;
25         if (rest % NOTE_SMALL == 0) {
26             *big = fives;
27             *small = rest / NOTE_SMALL;
28             found = 1;
29         }
30         fives--;
31     }
32     return found;
33 }
34
35 /* Returns 0 on success, -1 bad amount, -2 insufficient funds, -3 no notes */
36 static int withdraw(int amount)
37 {
38     int status = 0;
39     int big = 0;
40     int small = 0;
41
42     if (amount ≤ 0) {
43         status = -1;
```

```
44     } else if (amount > balance) {
45         status = -2;
46     } else if (!split_notes(amount, &big, &small)) {
47         status = -3;
48     } else {
49         balance -= amount;
50         printf("Paid %d as %d x %d and %d x %d\n",
51             amount, big, NOTE_BIG, small, NOTE_SMALL);
52     }
53     return status;
54 }
55
56 static void deposit(int amount)
57 {
58     if (amount ≤ 0) {
59         printf("Invalid amount\n");
60     } else {
61         balance += amount;
62         printf("Deposited %d\n", amount);
63     }
64 }
65
66 static void report_withdraw(int result)
67 {
68     switch (result) {
69     case -1:
70         printf("Invalid amount\n");
71         break;
72     case -2:
73         printf("Insufficient balance\n");
74         break;
75     case -3:
76         printf("Amount cannot be paid in 500 and 200 notes\n");
77         break;
78     default:
79         break;
80     }
81 }
82
83 int main(void)
84 {
85     int running = 1;
86     int choice;
87     int amount;
```

```
88
89     while (running) {
90         printf("\n1 Deposit  2 Withdraw  3 Balance  4 Exit\nChoice: ");
91         if (scanf("%d", &choice) != 1) {
92             running = 0;
93         } else {
94             switch (choice) {
95                 case 1:
96                     printf("Amount: ");
97                     if (scanf("%d", &amount) != 1) {
98                         running = 0;
99                     } else {
100
101                         deposit(amount);
102
103                     }
104
105                     break;
106
107                 case 2:
108
109                     printf("Amount: ");
110
111                     if (scanf("%d", &amount) != 1) {
112
113                         running = 0;
114
115                     } else {
116
117                         report_withdraw(withdraw(amount));
118
119                     }
120
121                     break;
122
123                 case 3:
124
125                     printf("Balance: %d\n", balance);
126
127                     break;
128
129                 case 4:
130
131                     running = 0;
132                 }
133             }
134         }
135     }
```

```
11
6         break;
11
7         default:
11
8             printf("Invalid choice\n");
11
9             break;
12
10        }
12
11    }
12
12    }
12
13    printf("Bye\n");
12
14    return 0;
12
15 }
```

What changed and what did not

The prompts, messages, note-splitting rule and balance arithmetic are unchanged. The program is longer (125 lines against 86) because each job now lives in a named function with a comment, and because the `switch` has explicit `break` lines. Every function has exactly one `return`. There is no `goto` and no label. The only flags are `running` and `found`, and each is tested only in the `while` condition of the loop it controls.

Sample Run

■ Write in lab record

Both programs were compiled with `gcc -Wall` (no warnings) and run on the same input sequence: deposit 1500, withdraw 300, withdraw 700, show balance, withdraw 900, deposit -50, choice 5, exit.

```
1 $ gcc -Wall -o unstructured unstructured.c
2 $ gcc -Wall -o structured structured.c
3 $ printf '1\n1500\n2\n300\n2\n700\n3\n2\n900\n1\n-50\n5\n4\n' > input.txt
4 $ ./unstructured < input.txt > a.out
5 $ ./structured < input.txt > b.out
6 $ diff a.out b.out && echo IDENTICAL
7 IDENTICAL
```

Interactive run of either program with the same choices:

```
1 1 Deposit 2 Withdraw 3 Balance 4 Exit
2 Choice: 1
3 Amount: 1500
4 Deposited 1500
5
6 1 Deposit 2 Withdraw 3 Balance 4 Exit
7 Choice: 2
8 Amount: 300
9 Amount cannot be paid in 500 and 200 notes
10
11 1 Deposit 2 Withdraw 3 Balance 4 Exit
12 Choice: 2
13 Amount: 700
14 Paid 700 as 1 x 500 and 1 x 200
15
16 1 Deposit 2 Withdraw 3 Balance 4 Exit
17 Choice: 3
18 Balance: 800
19
20 1 Deposit 2 Withdraw 3 Balance 4 Exit
21 Choice: 2
22 Amount: 900
23 Insufficient balance
24
25 1 Deposit 2 Withdraw 3 Balance 4 Exit
26 Choice: 1
27 Amount: -50
28 Invalid amount
29
30 1 Deposit 2 Withdraw 3 Balance 4 Exit
31 Choice: 5
32 Invalid choice
33
34 1 Deposit 2 Withdraw 3 Balance 4 Exit
35 Choice: 4
36 Bye
```

Viva Questions

✗ Do not copy. Read for understanding and the viva

- **Q:** Name the three control structures of structured programming. **A:** Sequence, selection and iteration, each with one entry and one exit.
- **Q:** Who proved that these three are enough? **A:** Bohm and Jacopini (1966); Dijkstra's "Go To Statement Considered Harmful" (1968) argued for applying it.
- **Q:** Why is a `goto` out of a nested loop worse than a `goto` to the next line? **A:** It gives the loop a second exit, so the code after the loop cannot assume the loop condition became false, and the loop variables have two possible meanings.
- **Q:** Is `break` inside a `switch` a violation? **A:** Strictly it is a jump, but it is the only way to end a case in C and it always lands at the same join point, so it is accepted.
- **Q:** How did you remove the early returns from `withdraw()`? **A:** An `if-else` chain assigns a `status` variable and the function returns it once at the end.
- **Q:** How did you prove the rewrite is equivalent? **A:** Same compiler flags, same input file, `diff` of the two outputs is empty.
- **Q:** The structured version is 40 lines longer. Is that a cost? **A:** In line count yes; in reading time no, because each function can be understood alone and the menu loop is visible as a loop.
- **Q:** What is a flag variable and when is it acceptable? **A:** A variable whose only job is to steer control flow; it is acceptable when it is the condition of the loop it controls and is set nowhere else.

Common Mistakes

× Do not copy. Read for understanding and the viva

- Choosing code that has no violations and inventing some in the write-up. The manual says: if there are none, pick another portion.
- Listing "uses goto" as one violation. Each label and each jump breaks a specific rule; the table needs one row per construct with a line number.
- Rewriting by deleting the `goto` and leaving the logic broken. Run both versions on the same input and compare; if the outputs differ, the rewrite is wrong.
- Replacing `goto` with a flag that is set in five places and tested in one. That moves the spaghetti into data instead of removing it.
- Forgetting to explain why each construct violates sequence, selection or iteration. The table's third column is the marks.

Session Summary

■ Write in lab record

- The original code (`unstructured.c`) with line numbers.
- The violations table with line reference, construct, rule broken and rewrite for each row.
- The rewritten code (`structured.c`).
- The sample run showing identical output from both versions on the same input.

 [Edit this page](#)

[< Previous](#)
Session 15

Session 17 [Next >](#)