Chapter 4 - Program Control

Outline

4.1 Introduction
4.2 The Essentials of Repetition
4.3 Counter-Controlled Repetition
4.4 The For Repetition Structure
4.5 The For Structure: Notes and Observations
4.6 Examples Using the For Structure
4.7 The Switch Multiple-Selection Structure
4.8 The Do/While Repetition Structure
4.9 The `break` and `continue` Statements
4.10 Logical Operators
4.11 Confusing Equality (==) and Assignment (=) Operators
4.12 Structured Programming Summary

◀ ▶

4.1 Introduction

- This chapter introduces
  - Additional repetition control structures
    - **for**
    - **do/while**
  - **switch** multiple selection structure
  - **break** statement
    - Used for exiting immediately and rapidly from certain control structures
  - **continue** statement
    - Used for skipping the remainder of the body of a repetition structure and proceeding with the next iteration of the loop

4.2 The Essentials of Repetition

- Loop
  - Group of instructions computer executes repeatedly while some condition remains **`true`**
- Counter-controlled repetition
  - Definite repetition: know how many times loop will execute
  - Control variable used to count repetitions
- Sentinel-controlled repetition
  - Indefinite repetition
  - Used when number of repetitions not known
  - Sentinel value indicates "end of data"

4.3 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires
  - The name of a control variable (or loop counter)
  - The initial value of the control variable
  - A condition that tests for the final value of the control variable (i.e., whether looping should continue)
  - An increment (or decrement) by which the control variable is modified each time through the loop

◀ ▶

4.3 Essentials of Counter-Controlled Repetition

- Example:

```
int counter = 1;             // initialization
while ( counter <= 10 ) { // repetition condition
   printf( "%d\n", counter );
   ++counter;                // increment
}
```

  - The statement

    ```
    int counter = 1;
    ```

    - Names **counter**
    - Declares it to be an integer
    - Reserves space for it in memory
    - Sets it to an initial value of **1**

◀ ▶

4.4 The `for` Repetition Structure

- Format when using **`for`** loops

  `for` (*initialization*; *loopContinuationTest*; *increment*)

  statement

- Example:

```
for( int counter = 1; counter <= 10; counter++ )
   printf( "%d\n", counter );
```

  - Prints the integers from one to ten

No semicolon (`;`) after last expression

4.4 The `for` Repetition Structure

- For loops can usually be rewritten as while loops:

```
initialization;
while ( loopContinuationTest ) {
  statement;
  increment;
}
```

- Initialization and increment
  - Can be comma-separated lists
  - Example:

```
for (int i = 0, j = 0;  j + i <= 10; j++, i++)
   printf( "%d\n", j + i );
```

4.5 The `for` Structure: Notes and Observations

- Arithmetic expressions
  - Initialization, loop-continuation, and increment can contain arithmetic expressions. If `x` equals `2` and `y` equals `10`

```
for ( j = x; j <= 4 * x * y; j += y / x )
```

is equivalent to

```
for ( j = 2; j <= 80; j += 5 )
```

- Notes about the `for` structure:
  - "Increment" may be negative (decrement)
  - If the loop continuation condition is initially `false`
    - The body of the `for` structure is not performed
    - Control proceeds with the next statement after the `for` structure
  - Control variable
    - Often printed or used inside for body, but not necessary

```c
/* Fig. 4.5: fig04_05.c
   Summation with for */
#include <stdio.h>

int main()
{
   int sum = 0, number;

   for ( number = 2; number <= 100; number += 2 )
      sum += number;

   printf( "Sum is %d\n", sum );

   return 0;
}
```

```
Sum is 2550
```

Outline

1. Initialize variables

2. `for` repetition structure

Program Output

4.7 The `switch` Multiple-Selection Structure

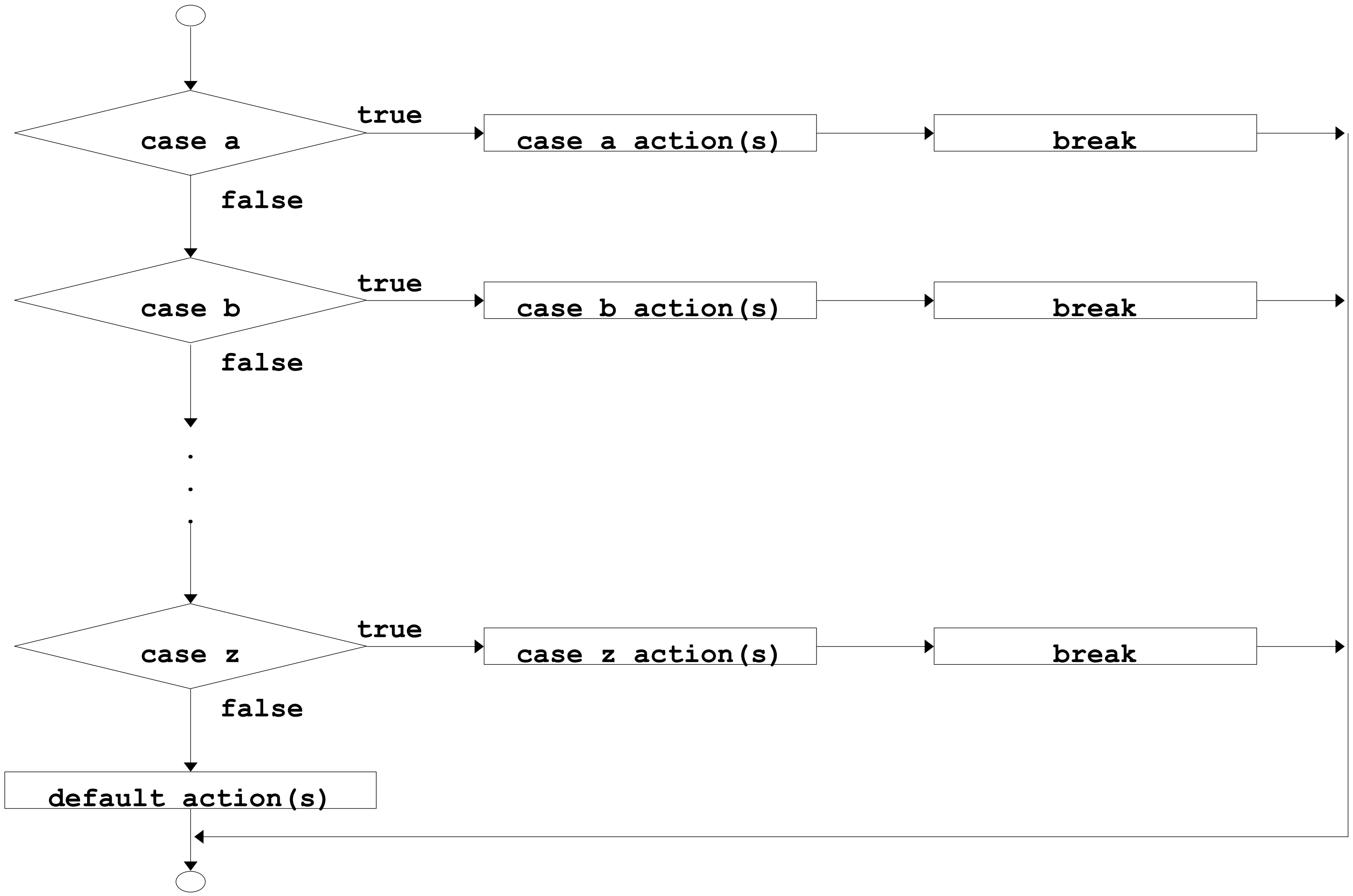
- **`switch`**
  - Useful when a variable or expression is tested for all the values it can assume and different actions are taken
- Format
  - Series of **`case`** labels and an optional **`default`** case

```
switch ( value ){
   case '1':
      actions
   case '2':
      actions
   default:
      actions
   }
```

  - **`break;`** exits from structure

4.7 The `switch` Multiple-Selection Structure

- Flowchart of the **`switch`** structure



```
/* Fig. 4.7: fig04 07.c
   Counting letter grades */
#include <stdio.h>

int main()
{
   int grade;
   int aCount = 0, bCount = 0, cCount = 0,
       dCount = 0, fCount = 0;

   printf(  "Enter the letter grades.\n"  );
   printf(  "Enter the EOF character to end input.\n"  );

   while ( ( grade = getchar() ) != EOF ) {

      switch ( grade ) {    /* switch nested in while */

         case 'A': case 'a':  /* grade was uppercase A */
            ++aCount;         /* or lowercase a */
            break;

         case 'B': case 'b':  /* grade was uppercase B */
            ++bCount;         /* or lowercase b */
            break;

         case 'C': case 'c':  /* grade was uppercase C */
            ++cCount;         /* or lowercase c */
            break;

         case 'D': case 'd':  /* grade was uppercase D */
            ++dCount;         /* or lowercase d */
            break;
```

Outline

1. Initialize variables

2. Input data

2.1 Use switch loop to update count

```
         case 'F': case 'f':  /* grade was uppercase F */
            ++fCount;         /* or lowercase f */
            break;

         case '\n': case' ':  /* ignore these in input */
            break;

         default:        /* catch all other characters */
            printf( "Incorrect letter grade entered." );
            printf( " Enter a new grade.\n" );
            break;
      }
   }

   printf( "\nTotals for each letter grade are:\n" );
   printf( "A: %d\n", aCount );
   printf( "B: %d\n", bCount );
   printf( "C: %d\n", cCount );
   printf( "D: %d\n", dCount );
   printf( "F: %d\n", fCount );

   return 0;
}
```

Outline

2.1 Use switch loop to update count

3. Print results

Program Output

```
Enter the letter grades.
Enter the EOF character to end input.
A
B
C
C
A
D
F
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
B

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

4.8 The do/while Repetition Structure

- The **do**/**while** repetition structure
  - Similar to the **while** structure
  - Condition for repetition tested after the body of the loop is performed
    - All actions are performed at least once
  - Format:

```
do {
   statement;
} while ( condition );
```

4.8 The do/while Repetition Structure

- Example (letting counter = 1):

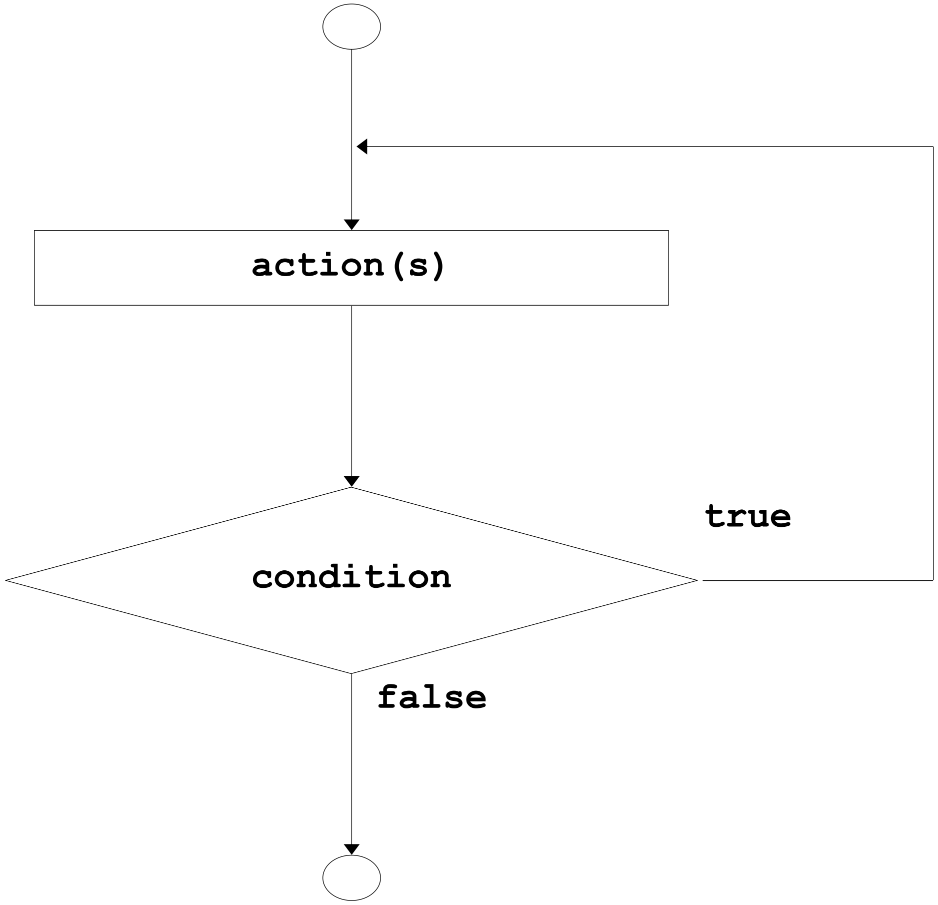
```
do {
   printf( "%d  ", counter );
} while (++counter <= 10);
```

  - Prints the integers from **1** to **10**

◀ ▶

4.8 The do/while Repetition Structure

- Flowchart of the **`do/while`** repetition structure



```c
/* Fig. 4.9: fig04_09.c
   Using the do/while repetition structure */
#include <stdio.h>

int main()
{
   int counter = 1;

   do {
      printf( "%d  ", counter );
   } while ( ++counter <= 10 );

   return 0;
}
```

1. Initialize variable

2. Loop

3. Print

```
1  2  3  4  5  6  7  8  9  10
```

Program Output

4.9 The `break` and `continue` Statements

- **`break`**
  - Causes immediate exit from a **`while`**, **`for`**, **`do`**/**`while`** or **`switch`** structure
  - Program execution continues with the first statement after the structure
  - Common uses of the **`break`** statement
    - Escape early from a loop
    - Skip the remainder of a **`switch`** structure

4.9 The break and continue Statements

- **continue**
  - Skips the remaining statements in the body of a **while**, **for** or **do**/**while** structure
    - Proceeds with the next iteration of the loop
  - **while** and **do**/**while**
    - Loop-continuation test is evaluated immediately after the **continue** statement is executed
  - **for**
    - Increment expression is executed, then the loop-continuation test is evaluated

```c
/* Fig. 4.12: fig04_12.c
   Using the continue statement in a for structure */
#include <stdio.h>

int main()
{
   int x;

   for ( x = 1; x <= 10; x++ ) {

      if ( x == 5 )
         continue;  /* skip remaining code in loop only
                       if x == 5 */

      printf( "%d ", x );
   }

   printf( "\nUsed continue to skip printing the value 5\n" );
   return 0;
}
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Outline

1. Initialize variable

2. Loop

3. Print

Program Output

4.10 Logical Operators

- **`&&`** (logical AND)
  - Returns **`true`** if both conditions are **`true`**
- **`||`** (logical OR)
  - Returns **`true`** if either of its conditions are **`true`**
- **`!`** (logical NOT, logical negation)
  - Reverses the truth/falsity of its condition
  - Unary operator, has one operand
- Useful as conditions in loops

Expression	Result
`true && false`	**`false`**
`true || false`	**`true`**
`!false`	**`true`**

4.11 Confusing Equality (==) and Assignment (=) Operators

- Dangerous error
  - Does not ordinarily cause syntax errors
  - Any expression that produces a value can be used in control structures
  - Nonzero values are **`true`**, zero values are **`false`**
  - Example using **`==`**:

```
if ( payCode == 4 )
   printf( "You get a bonus!\n" );
```

    - Checks **`paycode`**, if it is **`4`** then a bonus is awarded

4.11 Confusing Equality (==) and Assignment (=) Operators

- Example, replacing **==** with **=**:

```
    if ( payCode = 4 )
       printf( "You get a bonus!\n" );
```

  - This sets **paycode** to **4**
  - **4** is nonzero, so expression is **true**, and bonus awarded no matter what the **paycode** was

- Logic error, not a syntax error

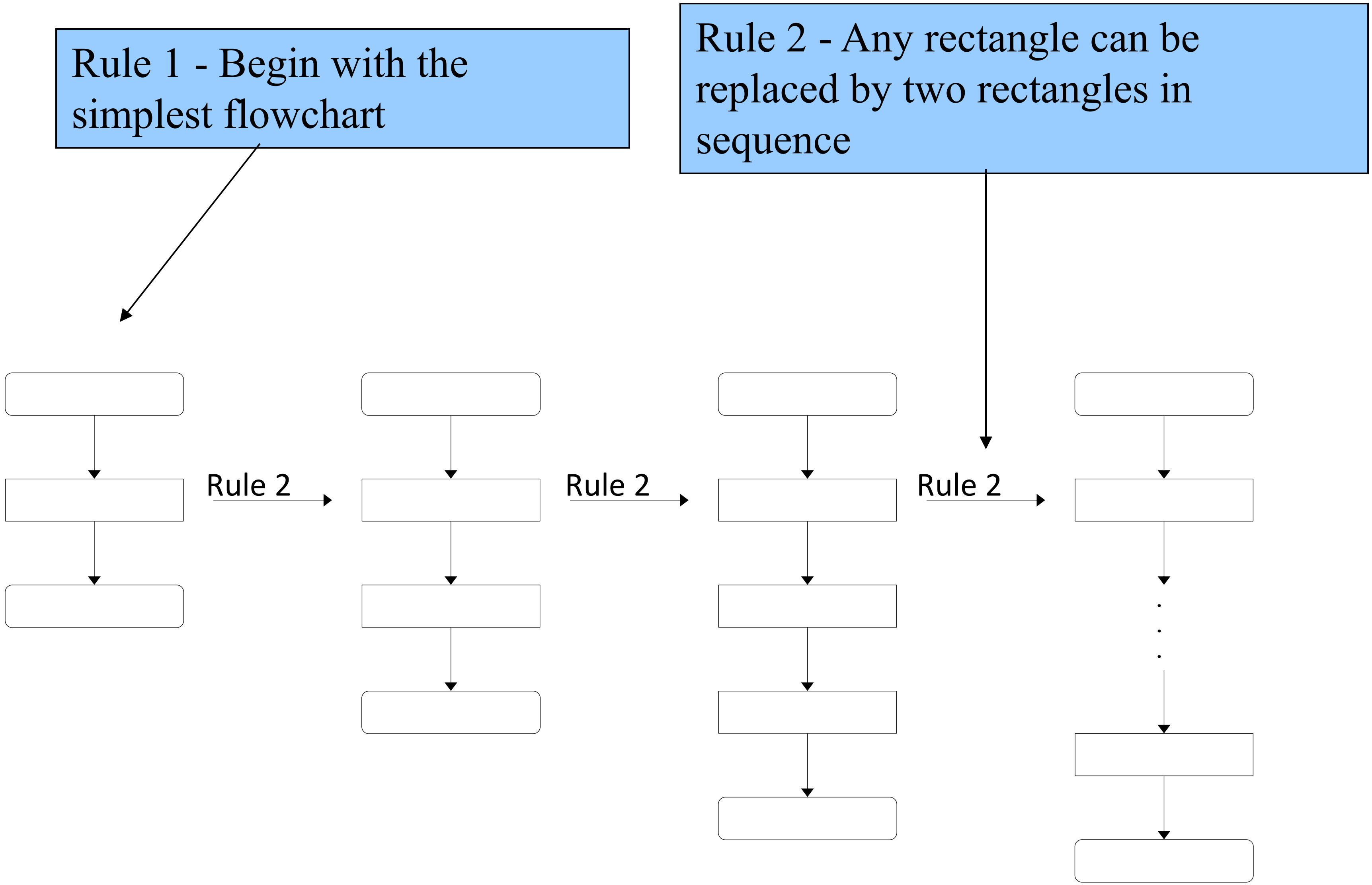
4.11 Confusing Equality (==) and Assignment (=) Operators

- lvalues
  - Expressions that can appear on the left side of an equation
  - Their values can be changed, such as variable names
    - `x = 4;`
- rvalues
  - Expressions that can only appear on the right side of an equation
  - Constants, such as numbers
    - Cannot write `4 = x;`
    - Must write `x = 4;`
  - lvalues can be used as rvalues, but not vice versa
    - `y = x;`

4.12 Structured-Programming Summary

- Structured programming
  - Easier than unstructured programs to understand, test, debug and, modify programs
- Rules for structured programming
  - Rules developed by programming community
  - Only single-entry/single-exit control structures are used
  - Rules:
    1. Begin with the “simplest flowchart”
    2. Any rectangle (action) can be replaced by two rectangles (actions) in sequence
    3. Any rectangle (action) can be replaced by any control structure (sequence, **`if`**, **`if`**/**`else`**, **`switch`**, **`while`**, **`do`**/**`while`** or **`for`**)
    4. Rules 2 and 3 can be applied in any order and multiple times

4.12 Structured-Programming Summary

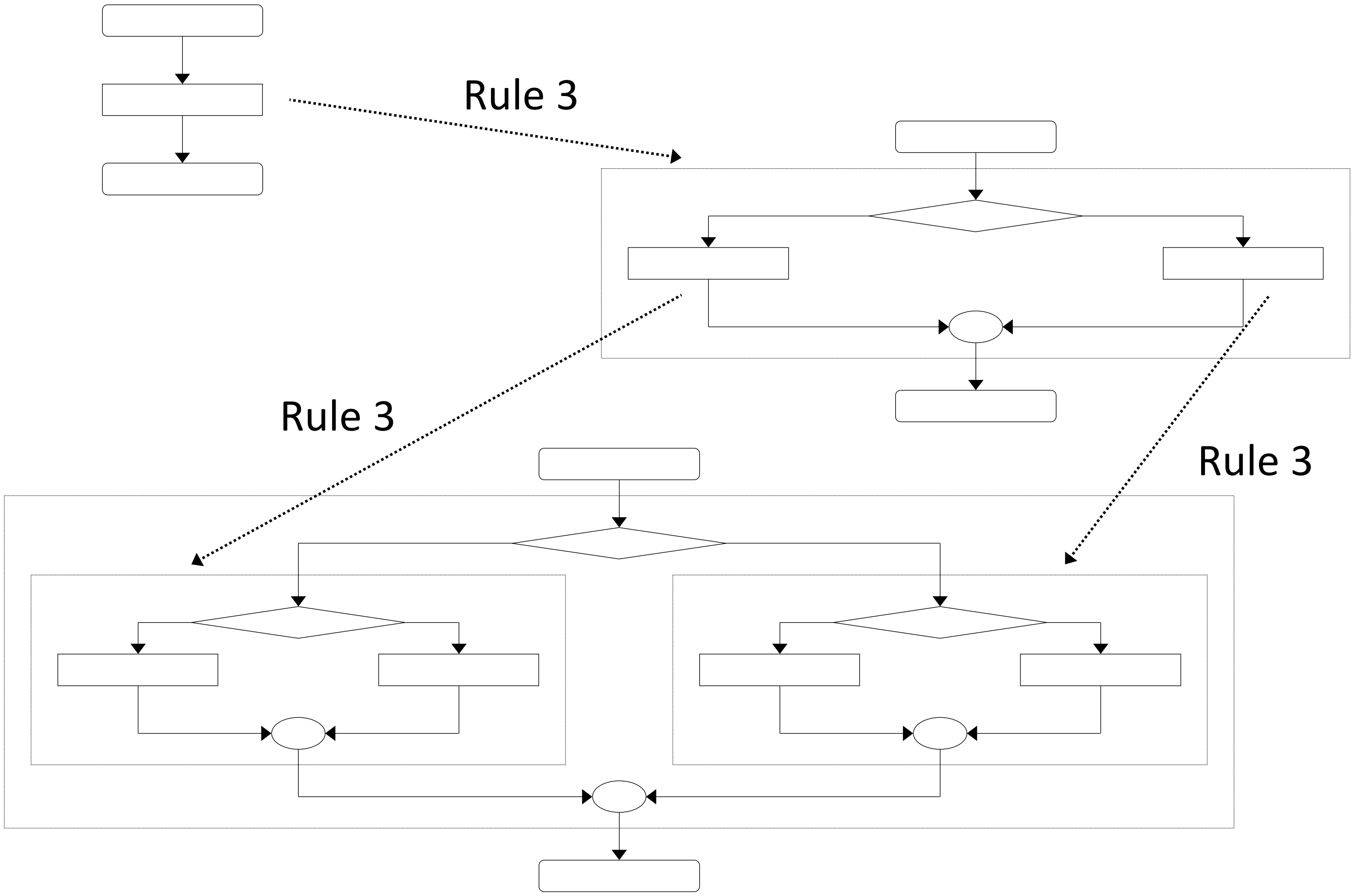


Rule 1 - Begin with the simplest flowchart

Rule 2 - Any rectangle can be replaced by two rectangles in sequence

4.12 Structured-Programming Summary

Rule 3 - Replace any rectangle with a control structure



4.12 Structured-Programming Summary

- All programs can be broken down into 3 controls
  - Sequence – handled automatically by compiler
  - Selection – **`if`**, **`if`**/**`else`** or **`switch`**
  - Repetition – **`while`**, **`do`**/**`while`** or **`for`**
    - Can only be combined in two ways
      - Nesting (rule 3)
      - Stacking (rule 2)
  - Any selection can be rewritten as an **`if`** statement, and any repetition can be rewritten as a **`while`** statement