

COSC1101 – Programming Fundamentals

Maham Khan

Lecture – 9&10

Switch Statement continued from the last Lecture

Decisions Using *switch*

- The control statement that allows us to make a decision from the number of choices is called a **switch**,
or
- Switch is also called **a switch-case-default**, since these three keywords go together to make up the control statement

Switch --- Features

- The integer expression following the keyword switch could be:
 - an integer constant like 1, 2 or 3, or
 - a character constant like 'a' or
 - an expression that evaluates to an integer like $2*4$
- The constant in each case must be different from all the others.
- The “do this” lines in the body of switch construct represent any valid C statements

Switch --- General Format

```
switch ( integer expression )  
{  
  case constant 1 :  
    do this ;  
  case constant 2 :  
    do this ;  
  case constant 3 :  
    do this ;  
  default :  
    do this ;  
}
```

Switch statement ---- incorrect

```
void main( )  
{  
int i = 2 ;  
    switch ( i )  
    {  
case 1 :  
        printf ( "I am in case 1 \n" ) ;  
case 2 :  
        printf ( "I am in case 2 \n" ) ;  
case 3 :  
        printf ( "I am in case 3 \n" ) ;  
default :  
        printf ( "I am in default \n" ) ;  
    }  
}
```

/ What is the out put? */*

Switch statement ---- incorrect output

- I am in case 2
- I am in case 3
- I am in default

Why ?

- Correct program would be

Switch statement ---- correct

```
void main( )  
{  
    int i = 2 ;  
  
    switch ( i )  
    {  
        case 1 :  
            printf ( "I am in case 1 \n" ) ;  
            break ;  
        case 2 :  
            printf ( "I am in case 2 \n" ) ;  
            break ;  
        case 3 :  
            printf ( "I am in case 3 \n" ) ;  
            break ;  
        default :  
            printf ( "I am in default \n" ) ;  
    }  
}
```

Output:
I am in case 2

In switch statement : Cases can be put in any order

```
void main( )
{
    int i = 22 ;
    switch ( i )
    {
        case 121 :
            printf ( "I am in case 121 \n" ) ;
            break ;
        case 7 :
            printf ( "I am in case 7 \n" ) ;
            break ;
        case 22 :
            printf ( "I am in case 22 \n" ) ;
            break ;
        default :
            printf ( "I am in default \n" ) ;
    }
}
```

Output:
I am in case 22

switch statement: using character as constants

```
void main()
```

```
{
```

```
char c = 'x' ;
```

```
    switch ( c )
```

```
    {
```

```
        case 'v' :
```

```
            printf ( "I am in case v \n" ) ;
```

```
            break ;
```

```
        case 'a' :
```

```
            printf ( "I am in case a \n" ) ;
```

```
            break ;
```

```
        case 'x' :
```

```
            printf ( "I am in case x \n" ) ;
```

```
            break ;
```

```
    default :
```

```
        printf ( "I am in default \n" ) ;
```

```
    }
```

```
}
```

Output: I am in x

Here 'v', 'a', 'x' are replaced by the ASCII values (118, 97, 120) of these character constants.

switch statement: combining multiple constants

```
void main( )
{
char ch ;
printf ( "Enter any of the alphabet a, b, or c " ) ;
scanf ( "%c", &ch ) ;

    switch ( ch )
    {
        case 'a' :
        case 'A' :
            printf ( "a as in ashar" ) ;
            break ;

        case 'b' :
        case 'B' :
            printf ( "b as in brain" ) ;
            break ;

        case 'c' :
        case 'C' :
            printf ( "c as in cookie" ) ;
            break ;

        default :
            printf ( "do you knew what are alphabets?" ) ;
    }
}
```

switch statement: Combining multiple constants

Explanation

- Here, if an alphabet a is entered the case 'a' is satisfied and since there are no statements to be executed in this case the control automatically reaches the next case i.e. case 'A' and executes all the statements in this case
- Even if there are multiple statements to be executed in each case there is no need to enclose them within a pair of braces (unlike if, and else).

switch statement: -- Compiler does not report an error

```
void main( )
{
int i, j ;
printf ( "Enter value of i" ) ;
scanf ( "%d", &i ) ;
switch ( i )
{
    printf ( "Hello" ) ;
    case 1 :
        j = 10 ;
        break ;
    case 2 :
        j = 20 ;
        break ;
}
}
```

- Every statement in a switch must belong to some case or the other. If a statement doesn't belong to any case the compiler won't report an error. However, the statement would never get executed. For example, in the following program the `printf()` will never be executed.

switch statement: -- Features

- If we have no default case, then the program simply falls through the entire switch and continues with the next instruction (if any,) that follows the closing brace of switch.
- Is switch a replacement for if?
Yes and No.
Yes,
because it offers a better way of writing programs as compared to if, and

switch statement: -- Features contd...

No

because in certain situations we are left with no choice but to use if. For example we do not have a case in a switch which looks like:

case $i \leq 20$:

- All that we can have after the case is an int constant or a char constant or an expression that evaluates to one of these constants. Even a float is not allowed.
- The advantage of switch over if is that it leads to a more structured program and the level of indentation is manageable, more so if there are multiple statements within each case of a switch.

switch statement: -- Features contd...

- The break statement when used in a switch takes the control outside the switch. However, use of continue will not take the control to the beginning of switch as one is likely to believe.
- In principle, a switch may occur within another, but in practice it is rarely done. Such statements would be called nested switch statements.

switch statement: -- using Expressions

We can check the value of any expression in a switch. Thus the following switch statements are legal:

- `switch (i + j * k)`
- `switch (23 + 45 % 4 * k)`
- `switch (a < 4 && b > 7)`

Expressions can also be used in cases provided they are constant expressions. Thus :

- The case `3 + 7` is correct,
- The case `a + b` is incorrect.

switch Versus if-else Ladder

- There are some things that you simply cannot do with a switch. These are:
- A float expression cannot be tested using a switch
- Cases can never have variable expressions (for example it is wrong to say case a +3 :)
- Multiple cases cannot use same expressions. Thus the following switch is illegal:

```
switch ( a )  
{  
  case 3 :  
    ...  
  case 1 + 2 :  
    ...  
}
```

Why use switch?

- When switch has so many limitations, why use it?
- Because, switch works faster than an equivalent if-else ladder. How come?
- The compiler generates a jump table for a switch during compilation. As a result, during execution it simply refers the jump table to decide which case should be executed.
- if-elses are slower because they are evaluated at execution time.
-
- A switch with 10 cases would work faster than an equivalent if-else ladder.
- If the if-else conditions are simple and few in number, then if-else would work out faster than the lookup mechanism of a switch.

Switch --- Summary

- When we need to choose one among number of alternatives, a switch statement is used.
- The switch keyword is followed by an integer or an expression that evaluates to an integer.
- The case keyword is followed by an integer or a character constant.
- The control falls through all the cases unless the break statement is given.
- The usage of the goto keyword should be avoided as it usually violates the normal flow of execution.

```

/* Four Function Calculator */
void main(void)
{
    float num1 = 1.0, num2 = 1.0;
    char op;

    while ( ! ( num1 == 0.0 & num2 == 0.0))
    {
        printf ( "Type number, operator, number \n");
        scanf ( "%f %c %f", &num1, &op, &num2);
        switch ( op )
        {
            case '+':
                printf ( " = %f", num1 + num2);
                break;

            case '-':
                printf ( " = %f", num1 - num2);
                break;

            case '*':
            case 'x':
                printf ( " = %f", num1 * num2);
                break;

```

```

            case '/':
            case '\\':
                printf ( " = %f", num1 / num2);
                break;

            case '-':
                printf ( " = %f", num1 - num2);
                break;

            default:
                printf ( " unknown operator");
        }

```

```

    }

```

The *goto* Keyword

`goto` keyword!

Is used to transfer the control or flow of execution of the program

The *goto* Keyword

- avoid `goto` keyword!
- reasons that programs become unreliable, and hard to debug.
- In a difficult programming situation it seems so easy to use a `goto` to take the control where you want. However, almost always, there is a more elegant way of writing the same program using `if`, `for`, `while` and `switch`. These constructs are far more logical and easy to understand.

The *goto* Keyword

Conclusion:

- You can always get the job done without goto.
- With good programming skills and practices goto should always be avoided.
- So donot use it if you can avoid it !!

The use of *goto* statement *not recommended*

```
int main () {
    unsigned Count;
    Count = rand () % 7;
    printf ("The count is %i. \nThis is ", Count);
        switch (Count) {
            case 0:
                printf ("none.\n");
                goto done;
            default:
                printf ("many.\n");
                goto done;
        }
done:
    return 0;
}
```

The use of *goto* statement *Recommended*

```
void main( ) {  
    int i, j, k ;  
    for ( i = 1 ; i <= 3 ; i++ )  
    {  
        for ( j = 1 ; j <= 3 ; j++ )  
        {  
            for ( k = 1 ; k <= 3 ; k++ )  
            {  
                if ( i == 3 && j == 3 && k == 3 )  
                    goto out ;  
                else  
                    printf ( "%d %d %d\n", i, j, k ) ;  
            }  
        }  
    }  
    out :  
    printf ( "Out of the loop at last!" ) ;  
}
```

The only programming situation in favor of using goto is when we want to take the control out of the loop that is contained in several other loops. The following program illustrates this:

Revision

- *control structure*
- *operators*
- *Structure programming*

switch Multiple-Selection Structure

- **switch**

- Test variable for multiple values
- Series of **case** labels and optional **default** case

```
switch ( variable ) {  
    case value1:           // taken if variable == value1  
        statements  
        break;             // necessary to exit switch  
  
    case value2:  
    case value3:           // taken if variable == value2 or == value3  
        statements  
        break;  
  
    default:               // taken if variable matches no other cases  
        statements  
        break;  
}
```

switch Multiple-Selection Structure

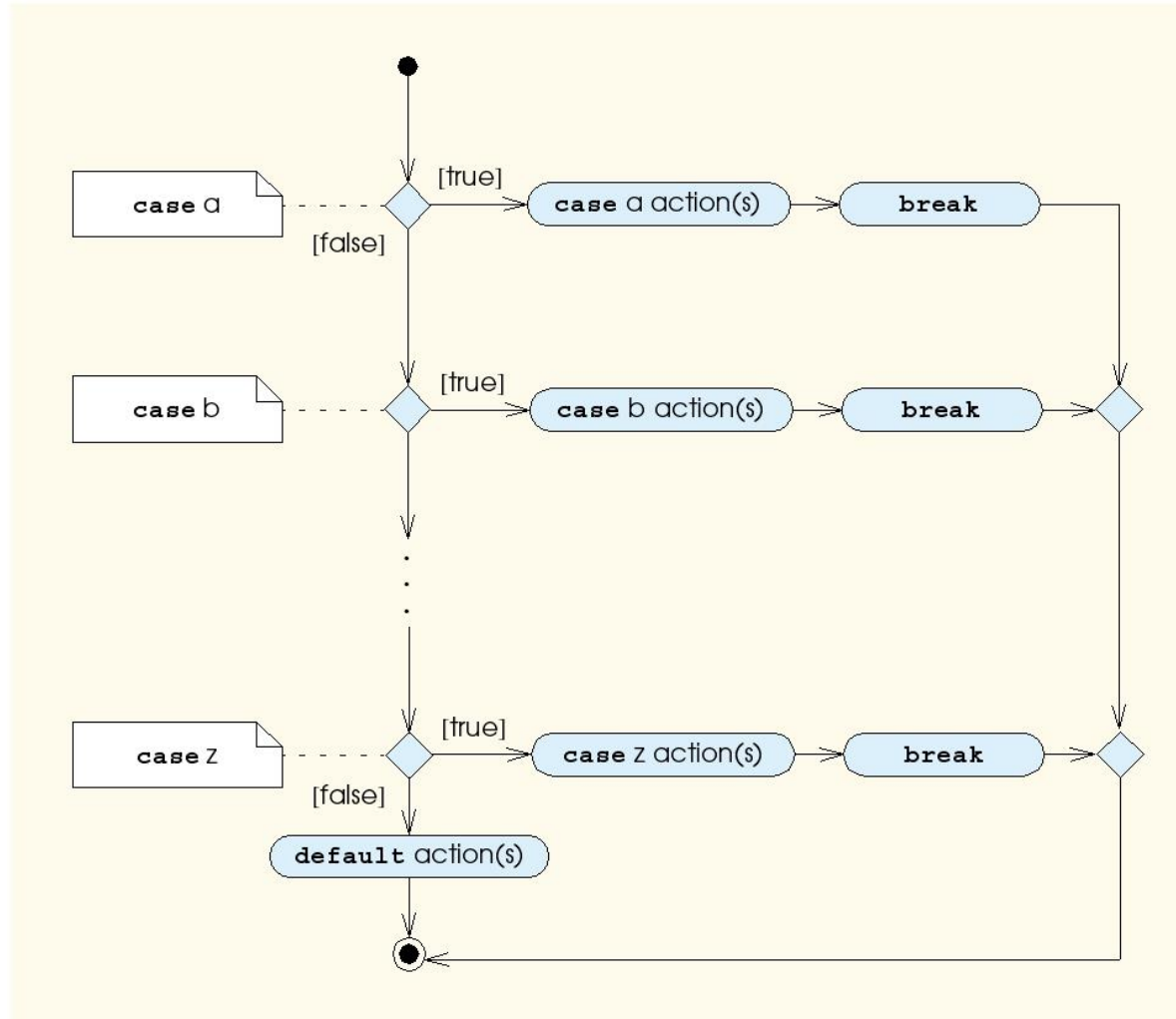


Fig. 2.23 **switch** multiple-selection structure activity diagram with **break** statements.

do/while Repetition Structure

- Similar to **while** structure
 - Makes loop continuation test at end, not in the beginning
 - Loop body executes at least once

- Format

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

do/while Repetition Structure

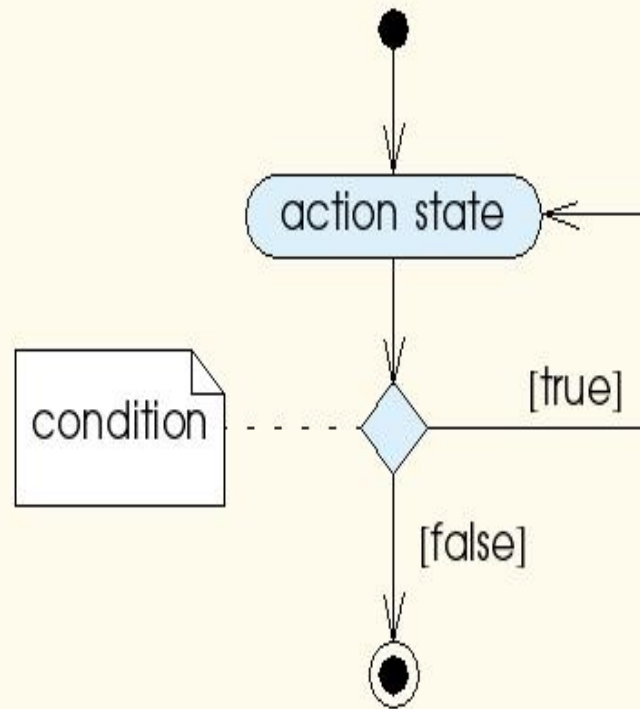


Fig. 2.25 **do/while** repetition structure activity diagram.

break and continue Statements

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

Example of Break

```
1 // Fig. 2.26: fig02_26.cpp
2 // Using the break statement in a for structure.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // function main begins program execution
9 int main()
10 {
11
12     int x; // x declared here so it can be used after the loop
13
14     // loop 10 times
15     for ( x = 1; x <= 10; x++ ) {
16
17         // if x is 5, terminate loop
18         if ( x == 5 )
19             break; // break loop only if x is 5
20
21         cout << x << " "; // display value of x
22
23     } // end for
24
25     cout << "\nBroke out of loop when x became " << x << endl;
26
27     return 0; // indicate successful termination
28
29 } // end function main
```

Exits **for** structure when **break** executed.

Output

1 2 3 4

Broke out of loop when x became 5

break and continue Statements

- **continue** statement
 - Used in **while**, **for**, **do/while**
 - Skips remainder of loop body
 - Proceeds with next iteration of loop
- **while** and **do/while** structure
 - Loop-continuation test evaluated immediately after the **continue** statement
- **for** structure
 - Increment expression executed
 - Next, loop-continuation test evaluated

Example of continue

```
1 // Fig. 2.27: fig02_27.cpp
2 // Using the continue statement in a for structure.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 // function main begins program execution
9 int main()
10 {
11     // loop 10 times
12     for ( int x = 1; x <= 10; x++ ) {
13
14         // if x is 5, continue with next iteration of loop
15         if ( x == 5 )
16             continue;    // skip remaining code in loop body
17
18         cout << x << " "; // display value of x
19
20     } // end for structure
21
22     cout << "\nUsed continue to skip printing the value 5"
23         << endl;
24
25     return 0;    // indicate successful termination
```

Skips to next iteration of the loop.

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

Logical Operators

- Used as conditions in loops, if statements

- **&&** (logical **AND**)

- **true** if both conditions are **true**

```
if ( gender == 1 && age >= 65 )  
    ++seniorFemales;
```

- **||** (logical **OR**)

- **true** if either of condition is **true**

```
if ( semesterAverage >= 90 || finalExam >= 90 )  
    cout << "Student grade is A" << endl;
```

Logical Operators

- **!** (logical **NOT**, logical negation)
 - Returns **true** when its condition is **false**, & vice versa

```
if ( !( grade == sentinelValue ) )  
    cout << "The next grade is " << grade << endl;
```

Alternative:

```
if ( grade != sentinelValue )  
    cout << "The next grade is " << grade << endl;
```

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

- Common error
 - Does not typically cause syntax errors
- Aspects of problem
 - Expressions that have a value can be used for decision
 - Zero = false, nonzero = true
 - Assignment statements produce a value (the value to be assigned)

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

- Example

```
if ( payCode == 4 )  
    cout << "You get a bonus!" << endl;
```

- If paycode is 4, bonus given

- If == was replaced with =

```
if ( payCode = 4 )  
    cout << "You get a bonus!" << endl;
```

- Paycode set to 4 (no matter what it was before)
- Statement is true (since 4 is non-zero)
- Bonus given in every case

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

- Lvalues
 - Expressions that can appear on left side of equation
 - Can be changed (i.e., variables)
 - **`x = 4;`**
- Rvalues
 - Only appear on right side of equation
 - Constants, such as numbers (i.e. cannot write **`4 = x;`**)
- Lvalues can be used as rvalues, but not vice versa

Structured-Programming Summary

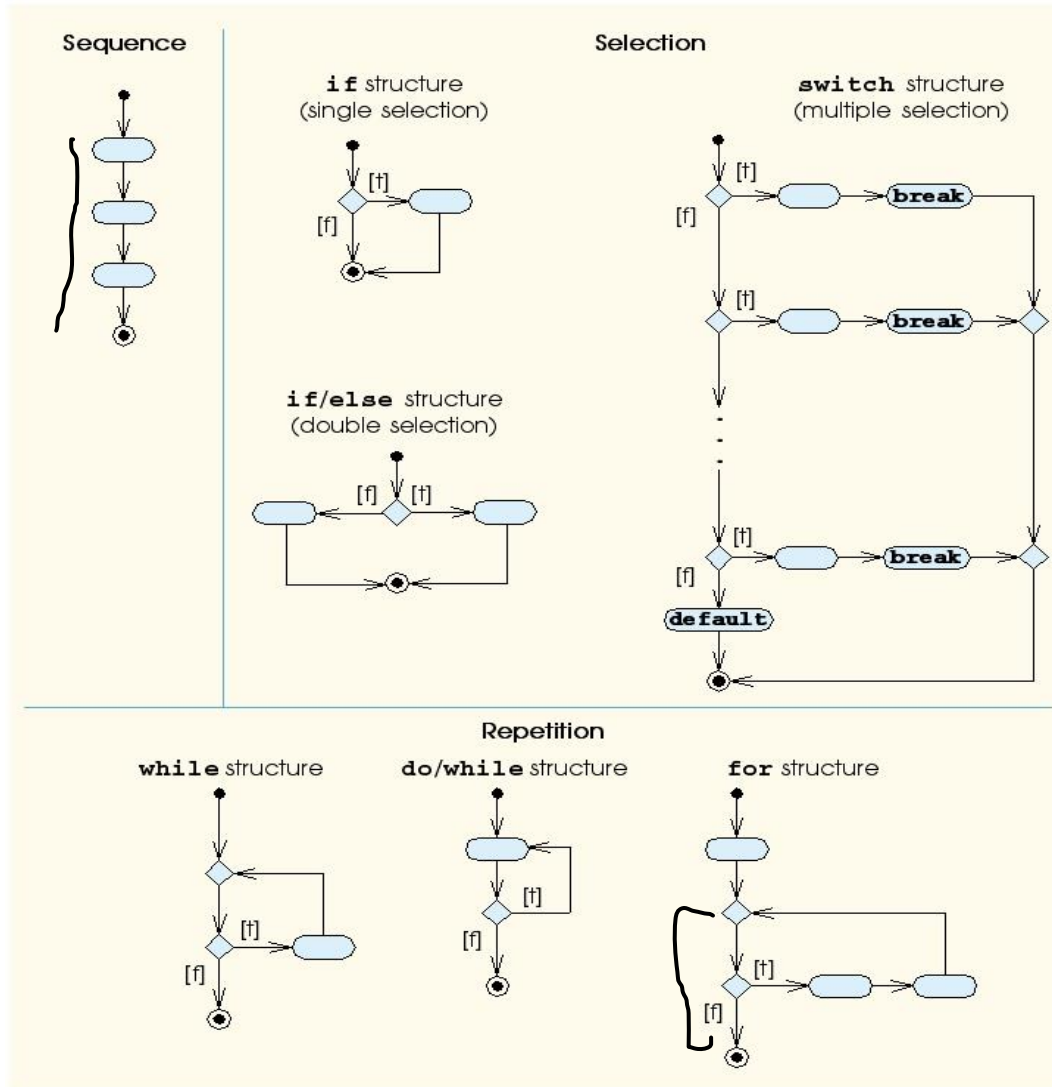


Fig. 2.32 C++'s single-entry/single-exit sequence, selection and repetition structures.

Structured-Programming Summary

- Structured programming
 - Programs easier to understand, test, debug and modify
- Rules for structured programming
 - Only use single-entry/single-exit control structures
 - 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

Structured-Programming Summary

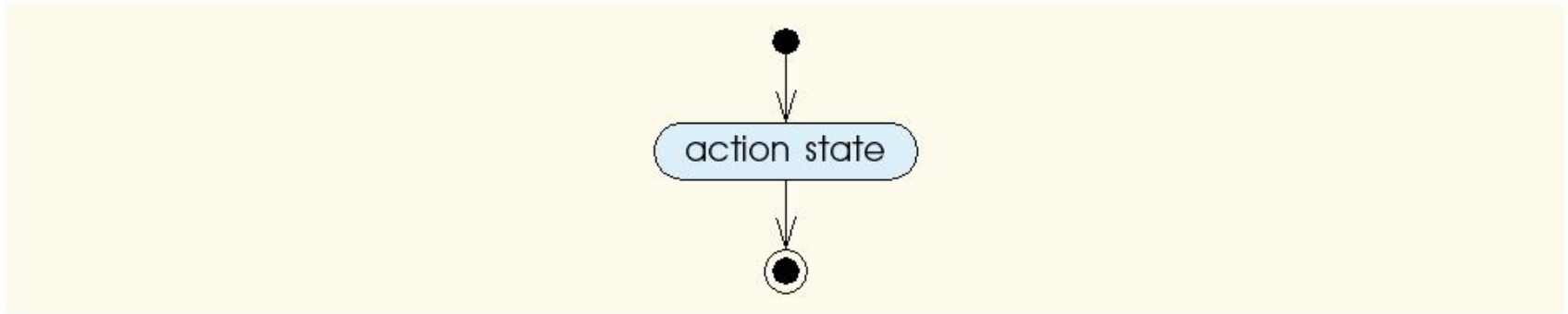


Fig. 2.34 Simplest activity diagram.

Structured-Programming Summary

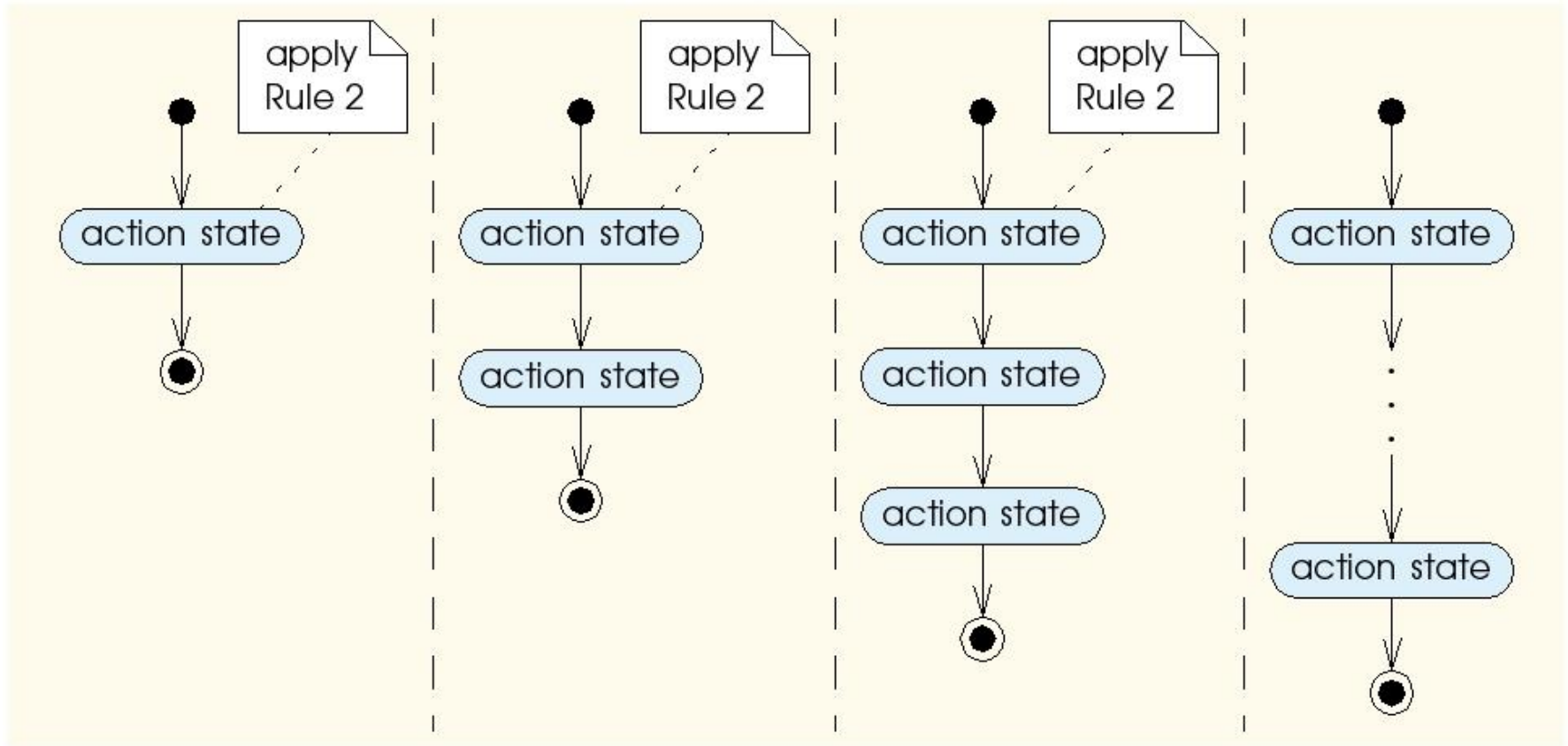


Fig. 2.35 Repeatedly applying rule 2 of Fig. 2.33 to the simplest activity diagram.

Structured-Programming Summary

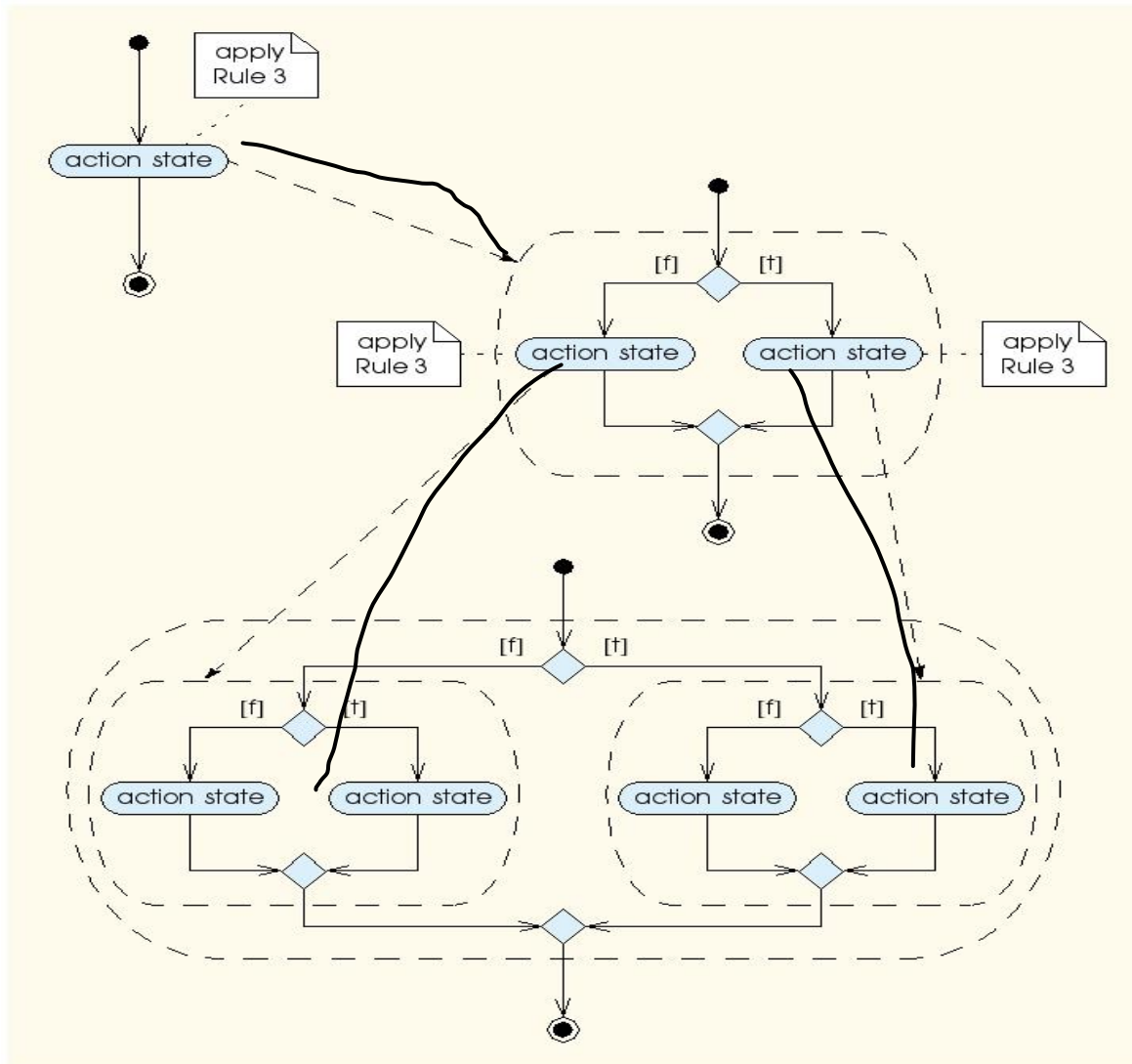


Fig. 2.36 Applying rule 3 of Fig. 2.33 to the simplest activity diagram.

Structured-Programming Summary

- All programs broken down into
 - Sequence
 - Selection
 - **if**, **if/else**, or **switch**
 - Any selection can be rewritten as an **if** statement
 - Repetition
 - **while**, **do/while** or **for**
 - Any repetition structure can be rewritten as a **while** statement