ENG1008 C Programming

Topic 3

❖ C Program Control

Objectives

- Review counter-controlled repetition
- To use the **`for`** and **`do…while`** repetition statements to execute statements repeatedly
- To use **`switch`** *selection statement* for multiple selection
- To use **`break`** and **`continue`** statements to change the program flow
- To use *logical operators* to build complicated expressions in conditional statements

Repetition Essentials

- A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true

- We have looked at two means of repetition:
  - Counter-controlled repetition
  - Sentinel-controlled repetition

- **Counter-controlled** repetition is sometimes called definite repetition because we know in advance exactly how many times the loop will be executed.
  - a control variable is used to *count* the *number of repetitions*
  - the *control variable* is *incremented* (normally 1) each time it goes through the loop
  - the *value* of the control variable helps to determine whether the *number of required repetitions* has been done
  - the *loop* then *terminates* and the execution continues with the statement after the repetition statements

Repetition Essentials

- **Sentinel-controlled** repetition is sometimes called indefinite repetition because it's not known in advance how many times the loop will be executed
    - **Sentinel value** are used to control the repetition
    - The precise *number of repetitions* are *not known in advance*
    - The loop contains statements that *obtains data in every loop*
    - The *sentinel* values indicates the *end of data*
        - It is *entered* after all (normal) data have been updated
        - It must be *different/distinct* from regular data

Counter-Controlled Repetition

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

Counter-Controlled Repetition

- The loop continuation condition in the while statement tests whether the control of the control variable is less than or equal to 10
- The body of the while loop is executed when the control variable is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- The loop terminates when the control variable exceeds 10 (i.e. when counter is 11)
- Another option is to loop when the control variable is 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (i.e. counter is initialized to 0 and the loop continuation condition in the while statement is less 10).

```c
// Fig. 4.1: fig04_01.c
// Counter-controlled repetition.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   unsigned int counter = 1; // initialization

   while ( counter <= 10 ) { // repetition condition
      printf ( "%u\n", counter ); // display counter
      ++counter; // increment
   } // end while
} // end function main
```

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

Fig. 4.1 | Counter-controlled repetition

Counter-Controlled Repetition

- The program could be made more concise by

```
while (++counter <= 10)
   printf("%u\n", counter);
```

  - This saves a statement as incrementing is done directly in the while condition before the condition is tested
  - This also eliminates the need for braces in the body of the while loop as it only contains one statement
  - This makes the code cryptic and may result in errors

Counter-Controlled Repetition

- **Floating-point counters**
  - Floating-point values are *approximate*
  - Controlling counting loops with floating–point variables may result in *imprecise counter values* and inaccurate termination test
  - Thus, we should only *control counting loops with integer values*
- Using **vertical spaces** and **indentation** – bodies of loops gives program a 2-dimensional look and more importantly it greatly *improves program readability*

❖ **For Repetition statement**

For Repetition statement

```c
// Fig. 4.2: fig04_02.c
// Counter-controlled repetition with the for statement.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   unsigned int counter; // define counter

   // initialization, repetition condition, and increment
   //  are all included in the for statement header.
   for ( counter = 1; counter <= 10; ++counter ) {
      printf( "%u\n", counter );
   } // end for
} // end function main
```

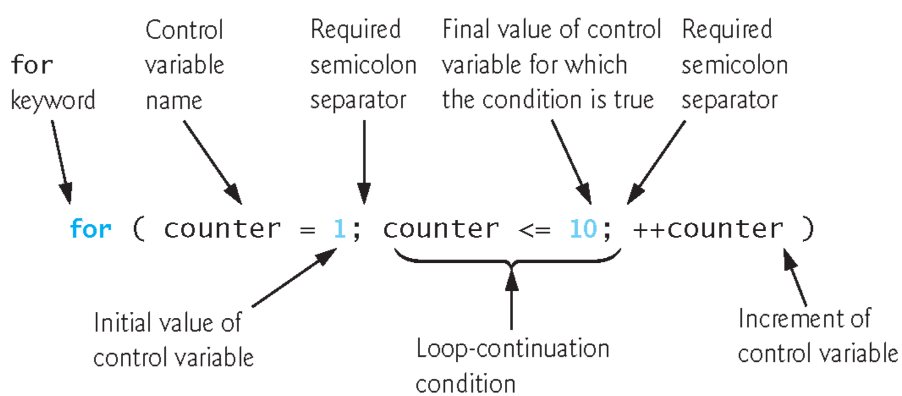

Fig. 4.3 | for statement header components.

- When the **for** loop starts, control variable **counter** is initialised to 1
- The loop-continuation condition **counter <= 10** is checked
- As the initial value of **counter** is 1, the condition above is satisfied
- The control variable **counter** is incremented by **++counter** and the loop begins again with the loop-continuation (counter is now 2)
- Braces are used to define the body of the **for** loop (if there are more than 1 statement)

This process continues until **counter** is incremented to 11 (loop continuation test fails and repetition terminates)

The program continues with the first statement after the **for** loop

For Repetition statement

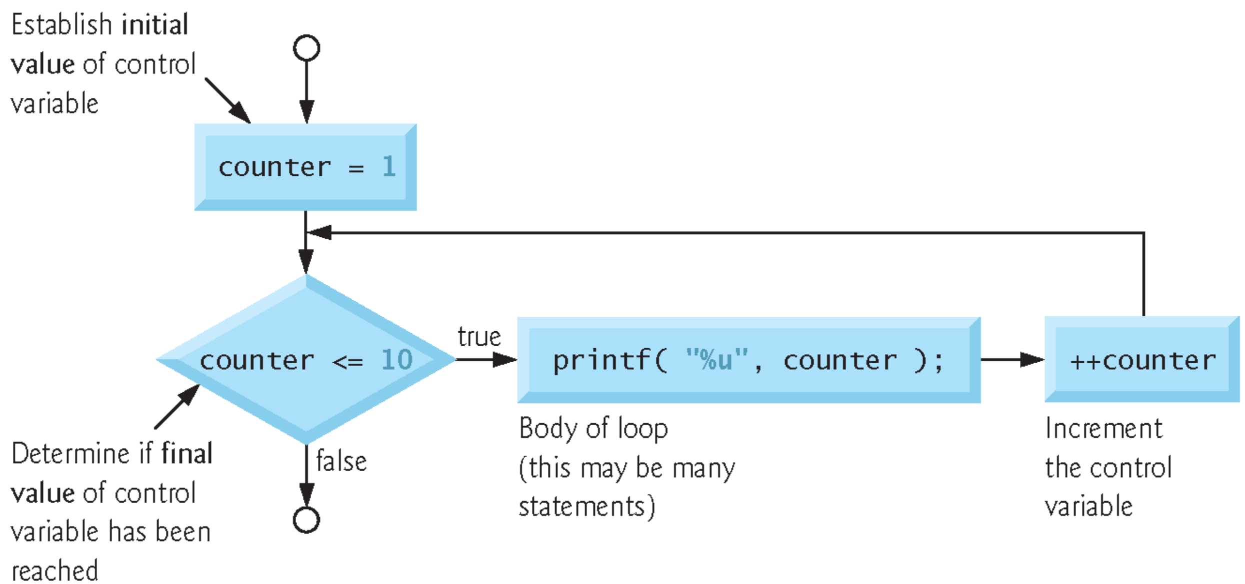

Fig. 4.4 | Flowcharting a typical `for` repetition statement.

- We could also work on

```
for (counter=0; counter<10; counter++)
```

whereby `counter` is initialized to 0 and the loop will be executed when `counter` ranges from 0 to 9; repetition terminates when `counter` is 10; the loop still executes 10 times

For Repetition statement

- **Off-by-one errors**
  - The loop may execute only 9 times
    - The *loop-continuation condition* counter is incorrectly written as **< 10** (instead of <= 10) and the initial value of `counter` is **1**
    - The loop will execute only 9 times
  - The loop may execute 11 times
    - The *loop-continuation condition* `counter` is incorrectly written as **<= 10** (instead of < 10) and the initial value of `counter` is **0**
    - The loop will execute 11 times
- Using *commas (,) instead of semicolons (;)* in the `for` header is a *syntax error*
- Increment expression below are all equivalent
  - `counter = counter + 1`
  - `counter += 1`
  - `++counter`
  - `counter++`

For Repetition statement

SINGAPORE INSTITUTE OF TECHNOLOGY

- General Format of **`for`** statement
  - The general format of the for statement is

```
for (expression1; expression2; expression3) {
    statement;
}
```

    where
    `expression1` initializes the loop-control variable,
    `expression2` is the loop-continuation condition, and
    `expression3` increments the control variable.

  - In most cases, the for statement can be represented with an equivalent while statement as follows:

```
expression1;
while (expression2) {
    statement;
    expression3;
}
```

  - There's an exception to this rule, which will be discussed later.

For Repetition statement

- General Format of **`for`** statement
  - Comma-separated lists of expressions
  - **`expression1`** and **`expression3`** are comma-separated lists of expressions
  - These are **comma operators** that guarantee that lists of expressions evaluate from left to right
  - The value and type of a comma-separated list of expressions are the value and type of the rightmost expression in the list
  - Its primary use is to enable you to use *multiple initialization* and/or *multiple increment expressions*
  - For example, there may be two control variables in a single for statement that must be initialized and incremented
    - e.g. `for (i=0, j=0; ...; i++, j++)`
  - Place only expressions involving the control variables in the initialization and increment section of a for statement

`for` Repetition statement

- Expressions in the `for` statement header are optional
  - The *three expressions* in the for statement are *optional*
  - If `expression2` is omitted, C assumes that the condition is true, thus creating an *infinite loop*
  - You may omit `expression1` if the control variable is *initialized elsewhere* in the program
  - `expression3` may be omitted if the *increment* is calculated by statements in the *body of the for statement* or if *no increment* is needed

- The value of the control variable *should not be changed in the body of a* `for` *loop (and also changed in the for header)* as this can lead to subtle errors

`for` Repetition statement

- The initialization, loop-continuation condition and increment can contain **arithmetic expressions**
- For example, if **`x = 2`** and **`y = 10`**, the statement

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

is equivalent to the statement

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

- The “increment” may be negative (in which case it’s really a decrement and the loop actually counts downward)
- If the *loop-continuation condition is initially false*, the loop body *does not execute*. Instead, execution proceeds with the statement following the `for` statement

For Repetition statement

- Examples using the `for` statement
  - Vary the control variable from 1 to 100 in increments of 1.
    `for (i=1; i<=100; i++)` or `for (i=1; i<=100; ++i)`
  - Vary the control variable from 100 to 1 in increments of -1 (decrements of 1).
    `for (i=100; i>= 1; i--)` or `for (i=100; i>= 1; --i)`
  - Vary the control variable from 7 to 77 in steps of 7.
    `for (i=7; i<= 77; i+=7)`
  - Vary the control variable from 20 to 2 in steps of -2.
    `for (i=20; i>= 2; i-=2)`
  - Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.
    `for (k=2; k<=17; k+=3)`
- The body of the for statement could be merged into the leftmost and rightmost portion of the for header by using comma operator
  `for (sum=0, number=2; number <= 100; sum+=number, number+= 2)`
  this is not recommended as it makes the program more difficult to read

For Repetition statement

- Application : Compound-Interest Calculations

- Consider the following problem statement:
  - A person invests $1000.00 in a savings account yielding 5% interest. Assuming that all interest is left on deposit in the account, **calculate and print the amount of money in the account at the end of each year for 10 years**.

- Use the following formula for determining these amounts:

$$a = p(1 + r)^n$$

where

p is the original amount invested (i.e., the principal)
r is the annual interest rate
n is the number of years
a is the amount on deposit at the end of the n^{th} year.

- This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit.

Sample output

Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.63
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

For Repetition statement

SINGAPORE INSTITUTE OF TECHNOLOGY

```c
1  // Fig. 4.6: fig04_06.c
2  // Calculating compound interest.
3  #include <stdio.h>
4  #include <math.h>
5
6  // function main begins program execution
7  int main( void )
8  {
9     double amount; // amount on deposit
10    double principal = 1000.0; // starting principal
11    double rate = .05; // annual interest rate
12    unsigned int year; // year counter
13
14    // output table column heads
15    printf( "%4s%21s\n", "Year", "Amount on deposit" );
16
17    // calculate amount on deposit for each of ten years
18    for ( year = 1; year <= 10; ++year ) {
19
20       // calculate new amount for specified year
21       amount = principal * pow( 1.0 + rate, year );
22
23       // output one table row
24       printf( "%4u%21.2f\n", year, amount );
25    } // end for
26 } // end function main
```

Year	Amount on deposit
1	1050.00
2	1102.50
3	1157.63
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

Fig. 4.6 | Calculating compound interest.

- The `for` statement executes the body of the loop 10 times, varying a control variable from 1 to 10 in increments of 1
- Although C does not include an exponentiation operator, we can use the Standard Library function `pow` for this purpose
- The function `pow(x,y)` calculates the value of x raised to the yth power
  - It takes two arguments of type **double** and returns a **double** value
  - Type **double** is a floating-point type like float, but typically a variable of type double can store a value of *much greater magnitude* with *greater precision* than float
- The header `<math.h>` (line 4) should be included whenever a math function such as `pow` is used
- Actually, this program would **malfunction** without the inclusion of `math.h`, as the linker would be unable to find the pow function
- The math.h file includes information that tells the compiler to convert the value of year to a temporary double representation *before* calling the function

- **Switch multiple-selection statement**

switch Multiple-selection statement

- **Multiple-selection** : there are a series of decisions in which a variable or expression is **tested separately** for each of the constant integral values it may assume, and **different actions** are taken
- C has the **`switch`** multiple-selection statement for such decision making
- The switch statement consists of a series of **`case`** **labels**, an optional **`default`** case and statements to execute for each case
- Figure 4.7 uses **`switch`** to count the number of each different letter grade students earned

```c
// Fig. 4.7: fig04_07.c
// Counting letter grades with switch.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   int grade; // one grade
   unsigned int aCount = 0; // number of As
   unsigned int bCount = 0; // number of Bs
   unsigned int cCount = 0; // number of Cs      initialization
   unsigned int dCount = 0; // number of Ds
   unsigned int fCount = 0; // number of Fs
   printf
   puts( "Enter the letter grades."  );
   puts( "Enter the EOF character to end input."  );

   // loop until user types end-of-file key sequence
   while ( ( grade = getchar() ) != EOF ) {
```

switch Multiple-selection statement

```
// determine which grade was input
switch ( grade ) { // switch nested in while
   case 'A': // grade was uppercase A
   case 'a': // or lowercase a
      ++aCount; // increment aCount
      break; // necessary to exit switch

   case 'B': // grade was uppercase B
   case 'b': // or lowercase b
      ++bCount; // increment bCount
      break; // exit switch

   case 'C': // grade was uppercase C
   case 'c': // or lowercase c
      ++cCount; // increment cCount
      break; // exit switch

   case 'D': // grade was uppercase D
   case 'd': // or lowercase d
      ++dCount; // increment dCount
      break; // exit switch

   case 'F': // grade was uppercase F
   case 'f': // or lowercase f
      ++fCount; // increment fCount
      break; // exit switch

   case '\n': // ignore newlines,
   case '\t': // tabs,
   case ' ': // and spaces in input
      break; // exit switch

   default: // catch all other characters
      printf( "%s", "Incorrect letter grade entered." );
      puts( " Enter a new grade." );
      break; // optional; will exit switch anyway
} // end switch
```

switch Multiple-selection statement

```c
    } // end while

    // output summary of results
    puts( "\nTotals for each letter grade are:" );
    printf( "A: %u\n", aCount ); // display number of A grades
    printf( "B: %u\n", bCount ); // display number of B grades
    printf( "C: %u\n", cCount ); // display number of C grades
    printf( "D: %u\n", dCount ); // display number of D grades
    printf( "F: %u\n", fCount ); // display number of F grades
} // end function main
```

```
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
^Z

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

Not all systems display a representation of the EOF character

Fig. 4.7 | Counting letter grades with switch. (Part 4 of 4.)

switch Multiple-selection statement

- In the program, the user enters letter grades for a class
  - In the `while` header (line 19),
    - `while ( ( grade = getchar() ) != EOF )`
  - the parenthesized assignment `(grade = getchar())` executes first
  - The `getchar` function from `<stdio.h>` reads one character from the keyboard and stores that character in the integer variable `grade`
  - Characters are normally stored in variables of type `char`
  - However, an important feature of C is that characters can be stored in any integer data type because they're usually represented as *one-byte* integers in the computer
  - The value of the assignment expression `grade = getchar()` is the character that's returned by `getchar` and assigned to the variable `grade`
  - In the program, the value of the assignment `grade = getchar()` is compared with the value of `EOF` (a symbol whose acronym stands for "end of file")

switch Multiple-selection statement

- `EOF`
  - We use `EOF` (which normally has the value `-1`) as the sentinel value
  - The user types a system-dependent keystroke combination to mean “end of file” - `EOF` is a symbolic integer constant defined in the `<stdio.h>` header
  - If the value assigned to grade is equal to `EOF`, the program terminates
  - We’ve chosen to represent characters in this program as int because `EOF` has an integer value (normally -1)
  - On *Linux/UNIX/Mac OS X* systems, the `EOF` indicator is entered by typing
    - `<Ctrl> d` on a line by itself
  - This notation `<Ctrl> d` means to press the `Enter` key then simultaneously press both `Ctrl` and `d`
  - On other systems, such as *Microsoft Windows*, the `EOF` indicator can be entered by typing
    - `<Ctrl> z`
  - You may also need to press `Enter` on *Windows*

switch Multiple-selection statement

- `switch`
  - `switch` is followed by the variable name `grade` in parentheses
  - This is called the controlling expression
  - The value of this expression is compared with each of the `case` labels
  - Assume the user has entered the letter C as a grade
    - C is automatically compared to each case in the `switch`
    - If a match occurs (`case 'C':`), the statements for that `case` are executed (in this case, `cCount` is incremented by 1)
    - the `switch` statement is *exited immediately* with the `break` *statement*
  - Each `case` can have *one or more actions*
  - The `switch` statement is different from all other control statements in that *braces are not required* around multiple actions in a `case` of a `switch`
  - The general `switch` multiple-selection statement (using a `break` in each `case`) is flowcharted in Fig. 4.8
  - The flowchart makes it clear that each `break` statement at the end of a `case` causes control to immediately exit the `switch` statement.

switch Multiple-selection statement

- `break`
  - causes program control to continue with the first statement after the `switch` statement
  - else the cases in a `switch` statement would otherwise run together
  - If `break` is not used anywhere in a `switch` statement, then each time a match occurs in the statement, the statements for all the remaining cases will be executed
- `default`
  - Cases not explicitly tested in a `switch` are ignored
  - The `default` case helps to collect these “un-tested” cases
  - Sometimes no `default` processing is required
  - Common to place the `default` clause last
  - `break` is not required for `default`; can be added for clarity

switch Multiple-selection statement

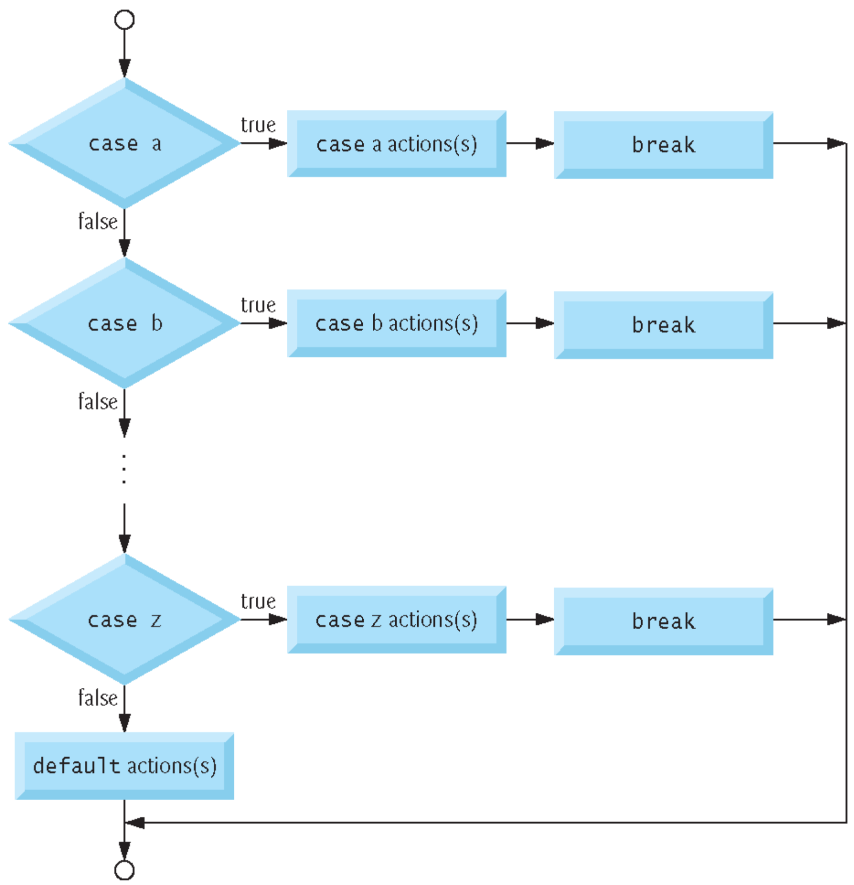

Fig. 4.8 | switch multiple-selection statement with breaks.

switch Multiple-selection statement

SINGAPORE INSTITUTE OF TECHNOLOGY

- ***Ignoring Newline, Tab and Blank Characters in Input***
  - In the `switch` statement of Fig. 4.7, the lines

```
case '\n': // ignore newlines,
case '\t': // tabs,
case ' ': // and spaces in input
   break; // exit switch
```

  - cause the program to skip newline, tab and blank characters
  - Reading characters one at a time can cause some problems
  - To have the program read the characters, you must send to the computer by pressing the `Enter` key
  - This causes the newline character to be placed in the input after the character we wish to process
  - Often, this newline character must be specially processed to make the program work correctly
  - By including the preceding cases in our `switch` statement, we prevent the error message in the default case from being printed each time a newline, tab or space is encountered in the input

switch Multiple-selection statement

- ***Listing several case labels together***

```
case 'D':
case 'd':
```

  simply means that the *same* set of actions is to occur for either of these cases

- ***Constant Integral Expressions***
  - each individual case can test only a constant integral expression—i.e., any combination of character constants and integer constants that evaluates to a constant integer value.
  - A character constant can be represented as the specific character in single quotes, such as 'A'
  - Characters *must* be enclosed within *single quotes* to be recognized as *character constants*
  - Characters in *double quotes* are recognized as *strings*
  - Integer constants are simply integer values
  - In our example, we have used character constants
  - Remember that characters are represented as small integer values

switch Multiple-selection statement

- ***Reading Character Input***
  - characters can be stored in any integer data type because they're usually represented as one-byte integers in the computer
  - we can treat a character as either an integer or a character, depends on its use
  - e.g. `printf( "The character (%c) has the value %d.\n", 'a', 'a' );`
  - uses the conversion specifiers `%c` and `%d` to print the character a and its integer value, respectively
  - The result is

    ```
    The character (a) has the value 97
    ```
  - Many computers today use the ASCII (American Standard Code for Information Interchange) character set in which 97 represents the lowercase letter 'a'
  - Characters can be read with `scanf` by using the conversion specifier `%c`

ASCII Table

```
Dec  = Decimal Value
Char = Character

'5' has the int value 53
if we write '5'-'0' it evaluates to 53-48, or the int 5
if we write char c = 'B'+32; then c stores 'b'
```

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

https://www.cs.cmu.edu/~pattis/15-1XX/common/handouts/ascii.html

SINGAPORE INSTITUTE OF TECHNOLOGY

- `Do...while` **Repetition statement**

do…while Repetition statement

- Similar to the **while** repetition statement
  - In the while loop, the loop-continuation condition is tested at the **start** of the loop before the body of the loop is executed

```
while (condition) {
      statements
}
```

- The **do…while** statement tests the loop-continuation condition **after** the loop body is executed

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

  - The do…while **loop body** is *executed at least once*
  - *Execution continues* with the statement after the while clause when the do…while loop terminates
  - *Braces* are not required if there's only one statement in the body
    - Usually included to avoid confusion with **while** loop

do...while Repetition statement

➢ Given the following code to print the numbers from 1 to 10

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

// function main begins program execution
int main( void )
{
   unsigned int counter = 1; // initialize counter

   do {
      printf( "%u  ", counter ); // display counter
   } while ( ++counter <= 10 ); // end do...while
} // end function main
```

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

Fig. 4.9 | Using the do...while repetition statement.

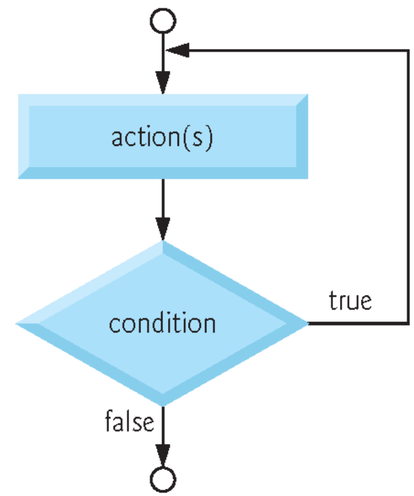

Fig. 4.10 | Flowcharting the do...while repetition statement.

- The control variable counter is *preincremented* (preferred) in the loop-continuation test (*for postincremented - "} while (counter++ < 10);"*)
- The flowchart in Fig 4.10 shows clearly the loop-continuation condition *does not* get *executed* until the *action* is *performed at least once*

Infinite Loops

- Caused when the loop-continuation condition in a repetition statement will *never become false*
- *Control variables* (e.g. counter) is *not incremented (or decremented) correctly* (or not at all) in the body of a counter-controlled repetition loop
- The *sentinel value* is *not entered* (at all) for a sentinel-controlled repetition loop
- There is a **semicolon immediately after a while or for** statement's header

```
for(; ;) {

}
while ;
```

while

do while

for

❖ break and continue statement

break and continue statements

- **break** and **continue** statements will change the *flow of control* of the program

- **break** statement
  - When executed in a *while, for, do…while,* or *switch* statement will result in an **immediate exit** from these statements
  - Program continues with the next instruction
  - Common use is to **escape** early from a loop or to **skip** the remainder of a switch statement

- In Figure 4.11, the break statement is used to terminate the *for loop earlier*
  - When x is equal to 5, the break statement is executed
  - How many times does the loop fully executes ?

break and continue statements

- Given the following code

```c
// Fig. 4.11: fig04_11.c
// Using the break statement in a for statement.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   unsigned int x; // counter

   // loop 10 times
   for ( x = 1; x <= 10; ++x ) {

      // if x is 5, terminate loop
      if ( x == 5 ) {
         break; // break loop only if x is 5
      } // end if

      printf( "%u ", x ); // display value of x
   } // end for

   printf( "\nBroke out of loop at x == %u\n", x );
} // end function main
```

```
1 2 3 4
Broke out of loop at x == 5
```

Fig. 4.11 | Using the break statement in a for statement.

break and continue statements

- **continue** statement
  - When executed in a *while, for, do…while,* or *switch* statement will **skip** the **remaining** statements in the body of the above statements and perform the **next iteration** of the loop
  - In a **while** and **do…while** loop, the *loop-continuation test* is *evaluated immediat*ely after the continue statement is executed
  - In a **for** loop, the *increment expression* is *executed first* before the *loop-continuation test* is *evaluated*
- In Chapter 4 Part 1 of the notes, it is noted that the **while** statement could be used to **represent** the **for** statement but with the exception:
  - when the *increment expression* in the body of the while loop occurs *after the continue statement*
  - the increment in the body of the while loop is not executed before the loop-continuation test
- In Figure 4.12, the continue statement in the *for* loop will skip the printf statement and begin the next iteration of the loop

break and continue statements

- Given the following code

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

// function main begins program execution
int main( void )
{
   unsigned int x; // counter

   // loop 10 times
   for ( x = 1; x <= 10; ++x ) {

      // if x is 5, continue with next iteration of loop
      if ( x == 5 ) {
         continue; // skip remaining code in loop body
      } // end if   break

      printf( "%u ", x ); // display value of x
   } // end for

   puts( "\nUsed continue to skip printing the value 5" );
} // end function main
```

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

Fig. 4.12 | Using the continue statement in a for statement.

break and continue statements

Software Engineering Observation 4.3

Some programmers feel that `break` and `continue` violate the norms of structured programming. The effects of these statements can be achieved by structured programming techniques we'll soon learn, so these programmers do not use `break` and `continue`.

Performance Tip 4.1

The `break` and `continue` statements, when used properly, perform faster than the corresponding structured techniques that we'll soon learn.

Software Engineering Observation 4.4

There's a tension between achieving quality software engineering and achieving the best-performing software. Often one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following guidelines: First, make your code simple and correct; then make it fast and small, but only if necessary.

❖ **Logical operators**

Logical Operators

- **C** provides *logical operators* that can be used to form *complex expressions / conditions*
- The logical operators are && (*logical AND*), || (*logical OR*) and ! (*logical NOT* also called *logical negation*).

- ***Logical AND (&&) operator***
  - Used to ensure that two simple conditions are true (instead of one) before we proceed with certain path of execution
  - if (gender == 1 **&&** age >= 65)
    seniorFemales++;
    - This **if** statement has 2 simple conditions
    - The 2 simple conditions are *evaluated first* (precedence rule)
    - The **if** statement then considers the *combined condition*
      gender == 1 **&&** age >= 65
    - The *combined condition* is *true* if *both* of the *simple conditions* are *true*
    - If *either or both* of the simple conditions are *false*, seniorFemales is not incremented and execution flows to the statement following the if

Logical Operators

- ***Logical AND (&&) operator***
  - C evaluates all expressions that include relational operators, equality operators, and/or logical operators to 0 or 1
  - C sets a true value to 1 but will accept any **nonzero value** as **true**

expression1	expression2	expression1 && expression2
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

Fig. 4.13 | Truth table for the logical AND (&&) operator.

Logical Operators

- ***Logical OR (||) operator***
  - Used to ensure that **either or both** of the two simple conditions are **true** before we proceed with certain path of execution
  - if (semesterAverage >= 90 **||** finalExam >= 90)

    printf("Student grade is A");
    - This **if** statement has 2 simple conditions
    - The 2 simple conditions are *evaluated first* (precedence rule)
    - The **if** statement then considers the *combined condition*

      semesterAverage >= 90 **||** finalExam >= 90
    - If *either or both* of the simple conditions are *true*, printf is executed
    - If *both* of the simple conditions are *false (zero)*, the message "Student grade is A" is not printed

expression1	expression2	expression1 || expression2
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Fig. 4.14 | Truth table for the logical OR (||) operator.

Logical Operators

SINGAPORE INSTITUTE OF TECHNOLOGY

- ***Logical Negation (!) operators***
  - **!** operator "reverses" the meaning of a condition (logical negation)
  - Unlike && and || which combine two conditions (binary operators), the logical negation operator operates on only one condition as an operand (unary operator)
  - Placed before a condition (to "reverse" the meaning)
  - if (**!**(grade == sentinelValue))
    printf("The next grade is %f\n", grade);
  - The parentheses around the condition "grade == sentinelValue" are needed because the logical negation operator has a higher precedence than the equality operator

expression	!expression
0	1
nonzero	0

Fig. 4.15 | Truth table for operator ! (logical negation).

SINGAPORE INSTITUTE OF TECHNOLOGY

➢ ***Summary of Operator Precedence and Associativity***

Operators	Associativity	Type
`++` *(postfix)* `--` *(postfix)*	right to left	postfix x--
`+` `-` `!` `++` *(prefix)* `--` *(prefix)* *(type)*	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=` true if (x>y) >3)	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 4.16 | Operator precedence and associativity.

Logical Operators

- ***Boolean type : _Bool Data Type***
  - boolean type
    - represented by the keyword _Bool
    - can hold only the values 0 or 1
  - Assigning any non-zero value to a _Bool will set it to 1
  - Include <stdbool.h> header
    - defines bool as a shorthand for type _Bool and
    - true and false as named representations of 1 and 0 respectively
  - At preprocessor time, bool, true and false are replaced with _Bool, 1 and 0

Confusing Equality (==) & Assignment (=)

- ***Accidental swapping of the operators == (equality) and = (assignment)***

  if (x = 1) ← always true

  if (x == 1) Comparison t/f

  - Normally will not result in compilation errors
    - which could be damaging
  - Program is able to compile and run but *incorrect results* may be generated due to *runtime logic errors*
  - 2 aspects of C cause these problems
    - Any expression in C that produces a value can be used in the decision portion of any control statement
    - Assignments in C produce a value (i.e. the value that is assigned to the variable on the left side of the assignment operator)

Confusing Equality (==) & Assignment (=)

SINGAPORE INSTITUTE OF TECHNOLOGY

- ***Accidental swapping of the operators == (equality) and = (assignment)***
  - For e.g.

    ```
    if (payCode == 4)
        printf("%s","You get a bonus!");
    if (payCode = 4)                        ← incorrectly written
        printf("%s","You get a bonus!");
    ```

    - payCode = 4 is a simple assignment whose value is 4;
      as nonzero value is interpreted as "true", the condition in this if statement is always true
  - For e.g. x = 1 was incorrectly written as x == 1
    - the value of x remains unaltered, probably causing an execution-time logic error

❖ **Structured Programming Summary**

Structured Programming Summary

- Figure 4.17 summarises the **control statements** (in Ch 3 and 4)
- Small **circles** are used in the figure to indicate the *single entry point* and *single exit point* of each statement
- Connecting control statements in sequence to form structured programs – the exit point of one control statement is connected directly to the entry point of the next
  - Control statements are simply placed one after another
  - Control-statement **stacking**
- Rules for structured programs also allow for control statements to be **nested**
- Figure 4.18 shows the rules for forming structured programs
  - Rectangle flowchart symbol can indicate any action including input and output (I/O)

Structured Programming Summary

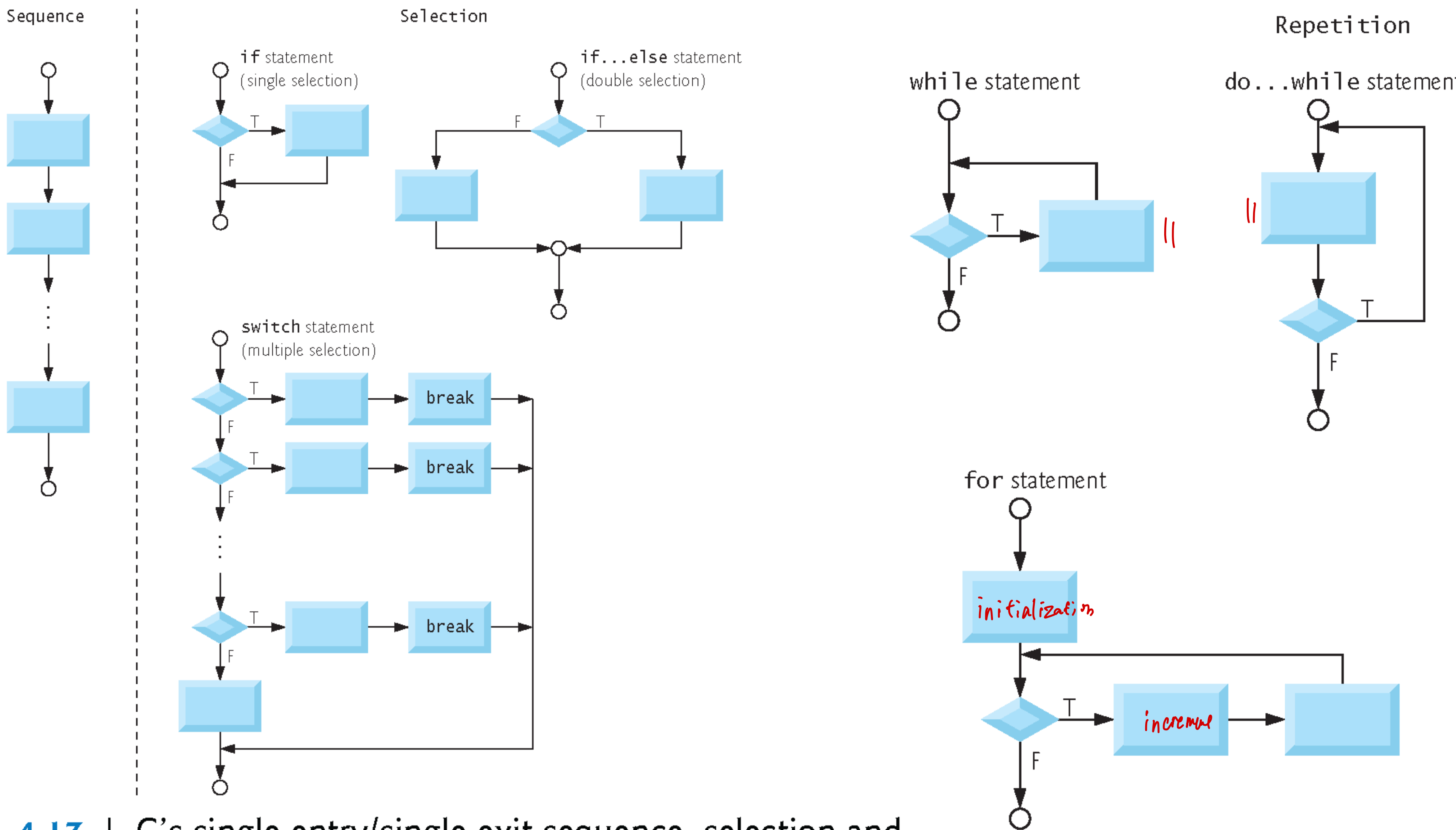

Fig. 4.17 | C's single-entry/single-exit sequence, selection and repetition statements

Structured Programming Summary

Rules for forming structured programs
1)Begin with the "simplest flowchart" (Fig. 4.19).
2)Any rectangle (action) can be replaced by *two* rectangles (actions) in sequence.
3)Any rectangle (action) can be replaced by *any* control statement (sequence, `if`, `if...else`, `switch`, `while`, `do...while` or `for`).
4)Rules 2 and 3 may be applied as often as you like and in *any* order.

Fig. 4.18 | Rules for forming structured programs.

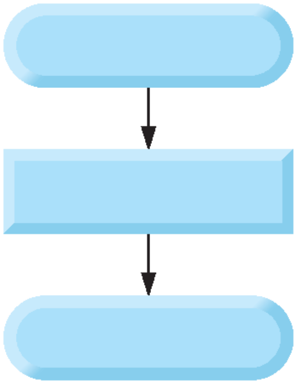

Fig. 4.19 | Simplest flowchart.

- Applying Fig 4.18 results in a structured flowchart with neat, building-block appearance
- Rule 2 generates a stack of control statements – ***stacking rule***
- Rule 3 results in a flowchart with neatly nested statements – ***nesting rule***
- Rule 4 generates larger, more involved, and more deeply nested structures

Structured Programming Summary

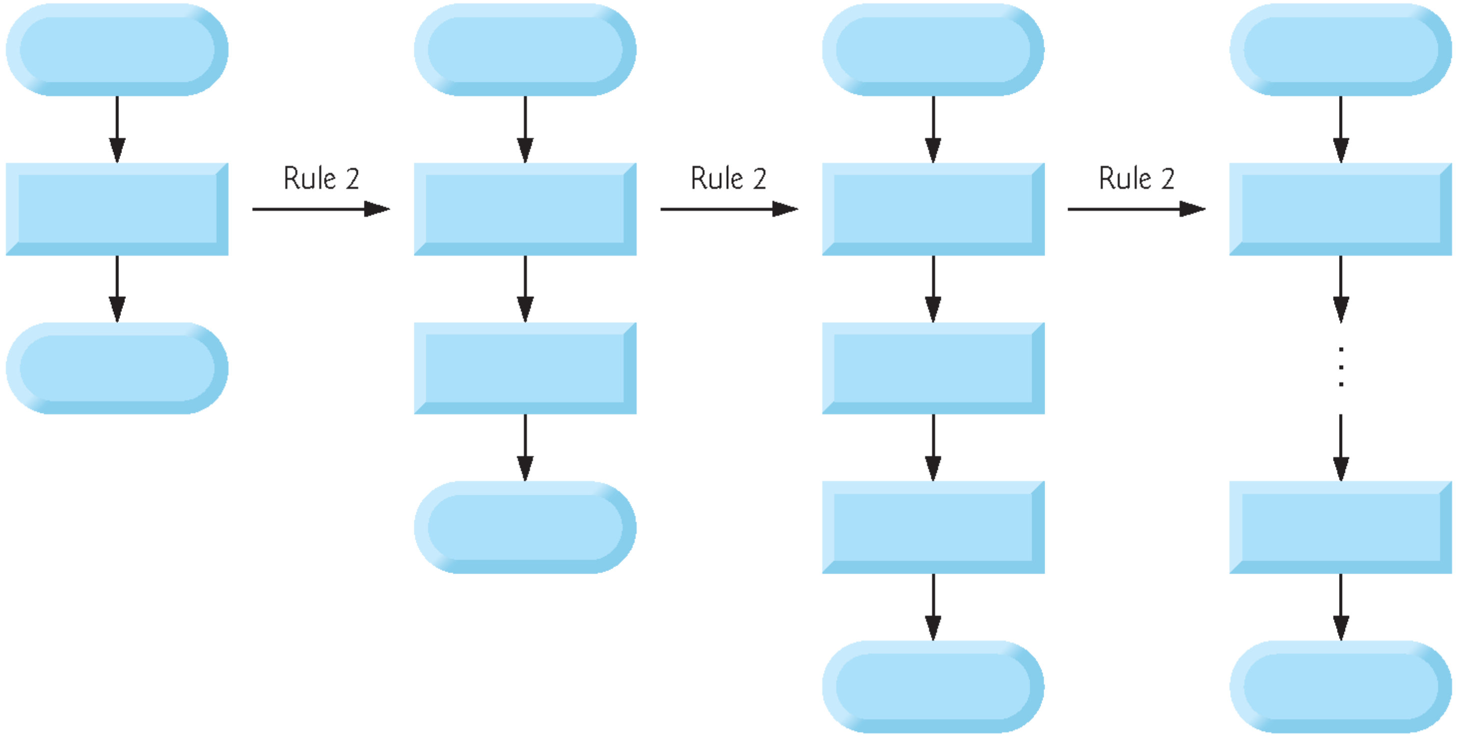

Fig. 4.20 | Repeatedly applying Rule 2 of Fig. 4.18 to the simplest flowchart.

Structured Programming Summary

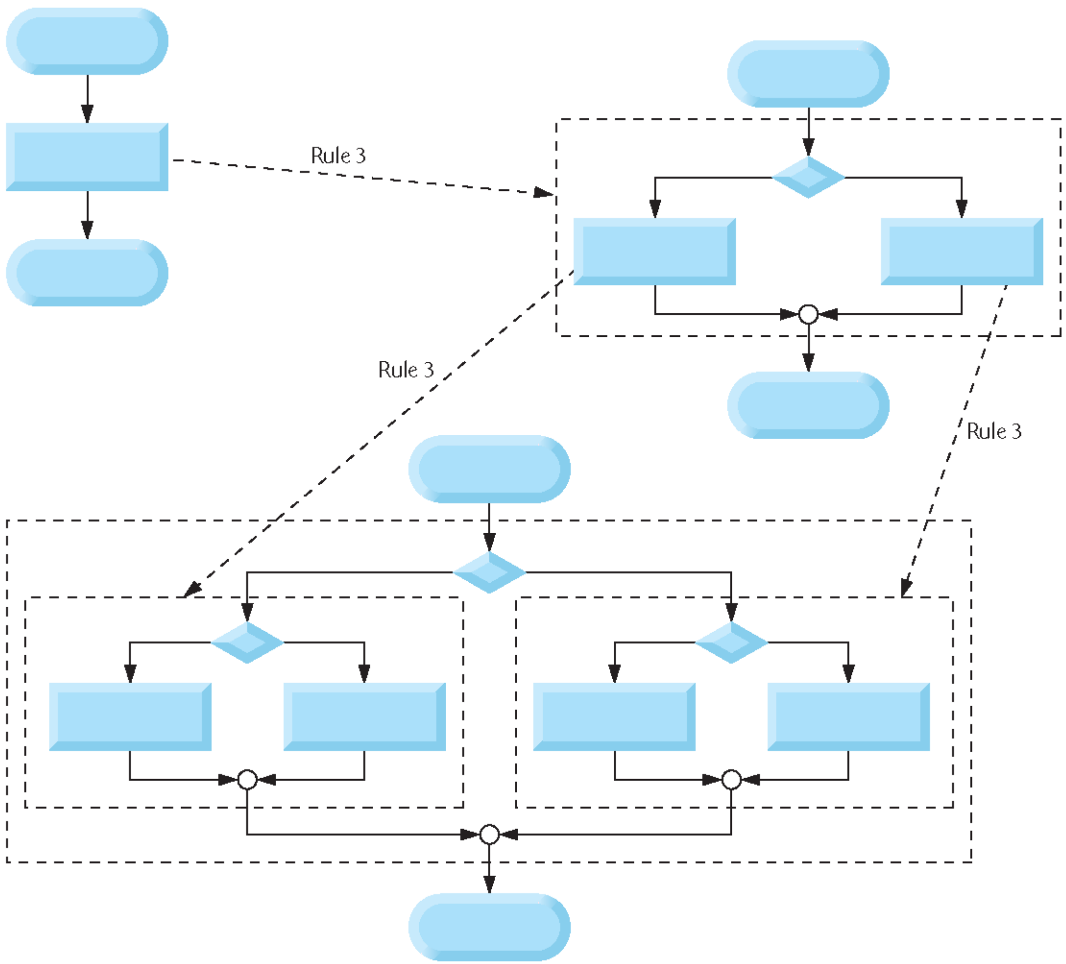

Fig. 4.21 | Applying Rule 3 of Fig. 4.18 to the simplest flowchart.

Structured Programming Summary

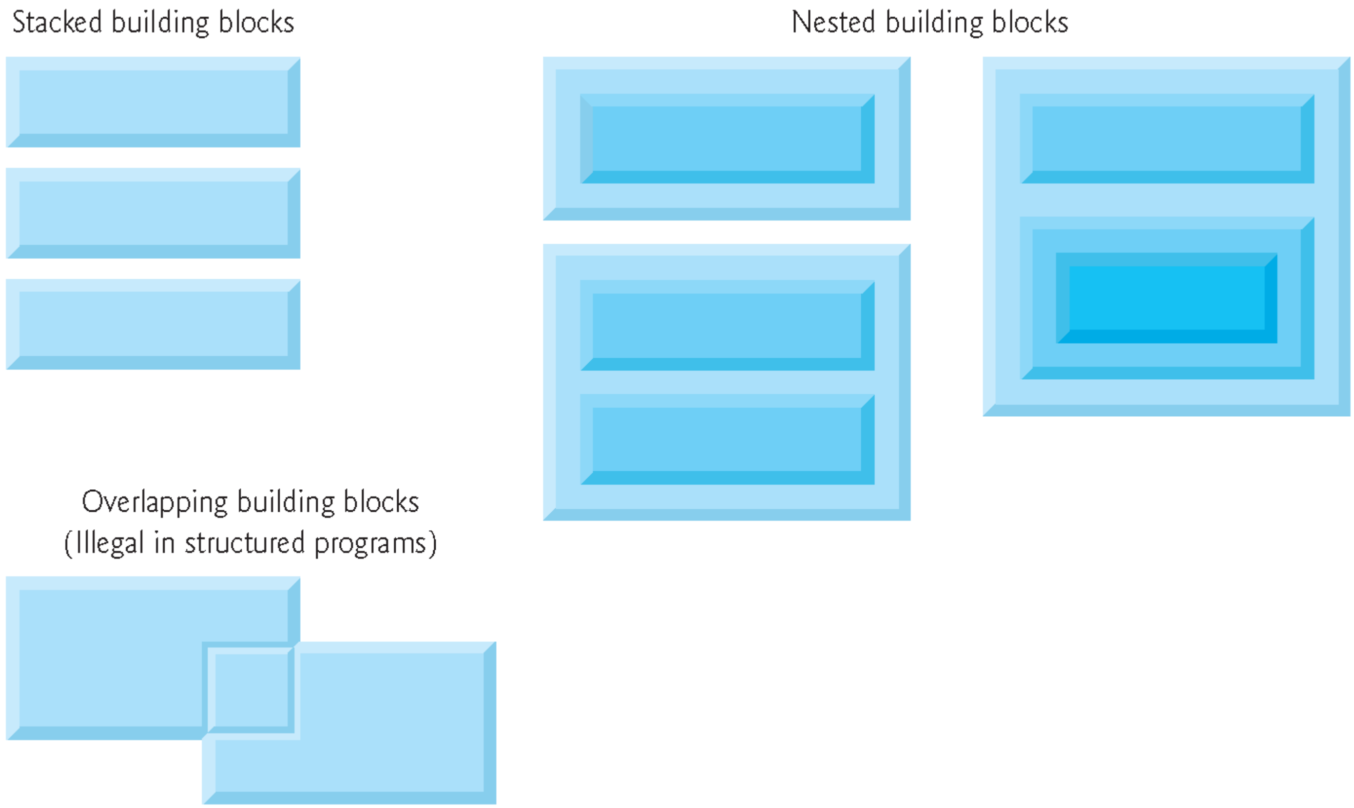

Fig. 4.22 | Stacked, nested and overlapped building blocks.

Structured Programming Summary

- **Sequence** is straightforward
- **Selection** is implemented in one of three ways:
  - if statement (single selection)
  - if…else statement (double selection)
  - switch statement (multiple selection)
  - The simple if statement is sufficient to give any form of selection
  - if…else and switch statements can be implemented with one or more if statements
- **Repetition** is implemented in one of three ways:
  - while statement
  - do…while statement
  - for statement

Structured Programming Summary

- The *while* statement is sufficient to provide any form of repetition
- Everything that can done with the *do…while* and *for* loops can be done with the *while* loop

➢ Thus, any form of control required in a C program can be expressed in terms of the following three forms of control:

- sequence
- if statement (selection)
- while statement (repetition)

➢ These control statements can be combined in only two ways – *stacking* and *nesting*