Chapter 4

C Program Control

Chapter 4 – C Program Control

Outline

4.1 Introduction
4.2 The Essentials of Repetition
4.3 Counter-Controlled Repetition
4.4 The for Repetition Statement
4.5 The for Statement: Notes and Observations
4.6 Examples Using the for Statement
4.7 The switch Multiple-Selection Statement
4.8 The do…while Repetition Statement
4.9 The break and continue Statements
4.10 Logical Operators
4.11 Confusing Equality (==) and Assignment (=) Operators

4.1 Introduction

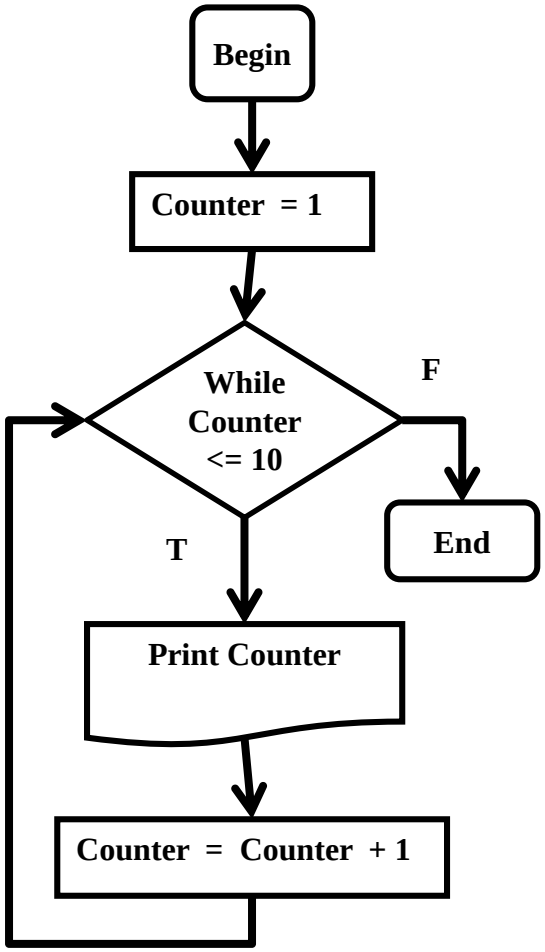
- This chapter introduces
  - Additional repetition (a.k.a. loop, iteration) control structures
    - `for`
    - `Do…while`
  - `switch` multiple selection statement
  - `break` statement
    - Used for exiting immediately and rapidly from certain control structures
  - `continue` statement
    - Used 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

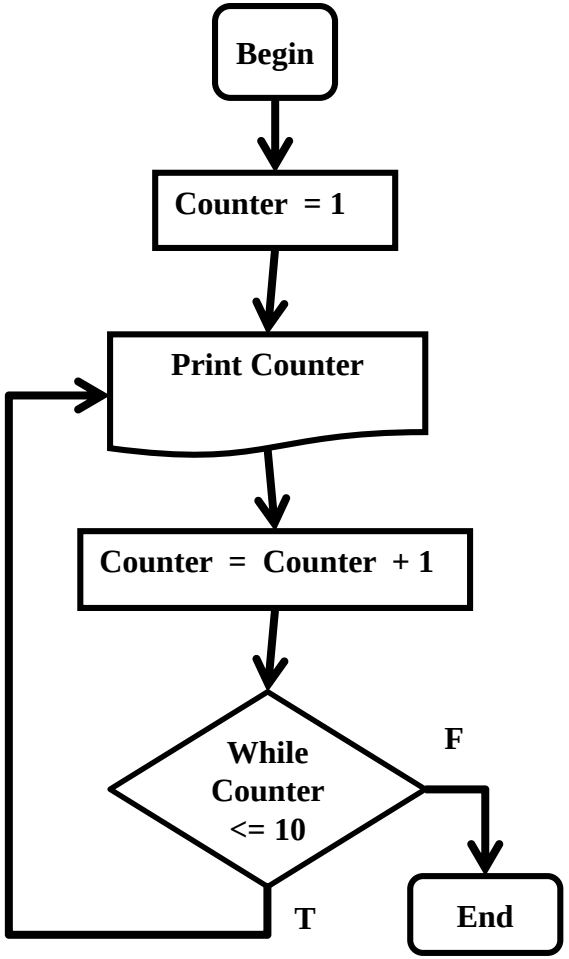
- Loop
  - Group of instructions computer executes repeatedly while some condition remains **`true`**
- Counter-controlled repetition
  - Definite repetition: know how many times loop will execute
  - Control variable used to count repetitions
- Sentinel-controlled repetition
  - Indefinite repetition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

Example: Flow Charts of repetitions (Counter-Controlled)

while

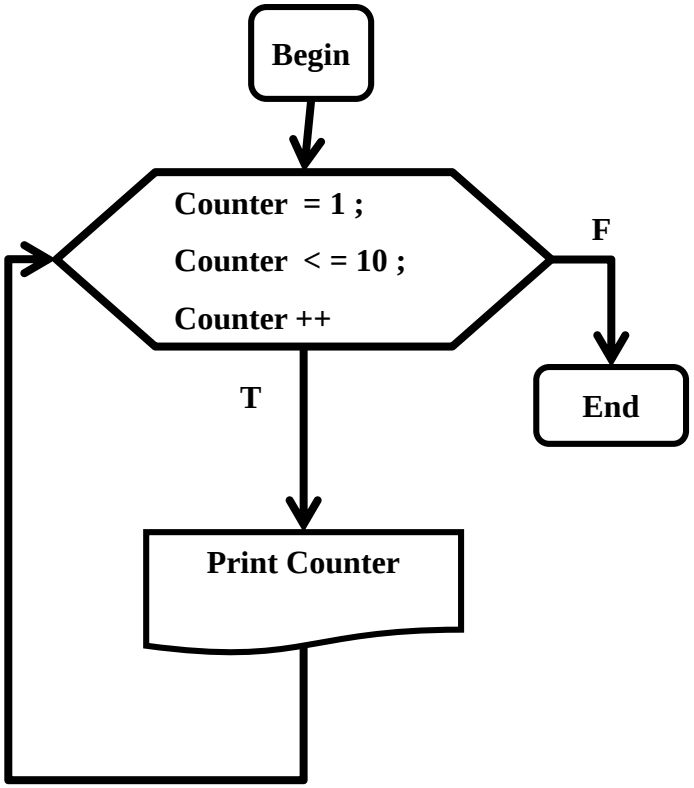


do .. while

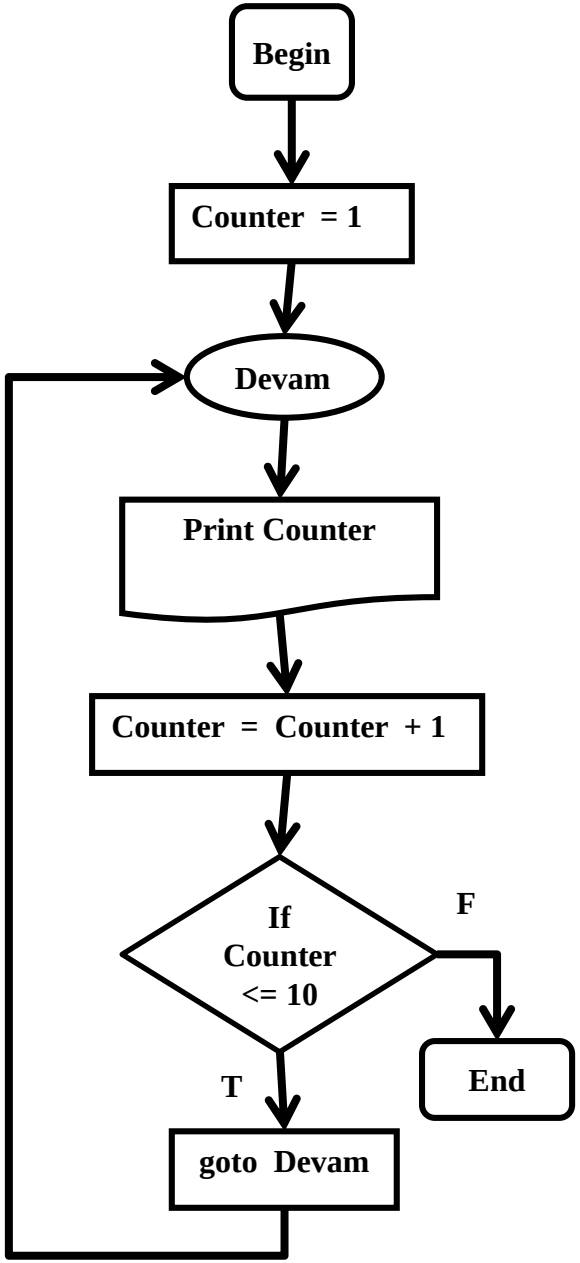


Example: Flow Charts of repetitions (Counter-Controlled)

for



goto



Example: C codes of repetitions (Counter-Controlled)

while

```c
#include <stdio.h>

int main()
{
   int counter = 1;

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

do .. while

```c
#include <stdio.h>

int main()
{
   int counter = 1;

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

Example: C codes of repetitions (Counter-Controlled)

for

```c
#include <stdio.h>

int main()
{
   int counter;

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

goto

```c
#include <stdio.h>

int main()
{
   int counter = 1;

 // This is a line label
 Devam:
   printf("%d\n", counter);
   counter++;

   if (counter <= 10)
      goto Devam;
}
```

Infinite Repetition

- When the termination condition of a repetition statement is coded with logical errors, the program will run inifinetely.
- Usually the operating system can not detect and stop an infinite program execution.

Program 1

```
#include <stdio.h>

int main()
{
  int a=1;

  while (a > 0) // infinite loop
  {
      printf("%d \n", a);
      a++;
  }
}
```

Program Output

```
1
2
3
4
...
...
...
...
...
...
(infinite)
```

Program 2

```c
#include <stdio.h>

int main()
{
  while (1) // infinite loop
      printf("Hello \n");
}
```

Program Output

```
Hello
Hello
Hello
Hello
...
...
...
(infinite)
```

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
  - An increment (or decrement) by which the control variable is modified each time through the loop
  - A condition that tests for the final value of the control variable (i.e., whether looping should continue)

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`
    - Defines it to be an integer
    - Reserves space for it in memory
    - Sets it to an initial value of `1`

```c
/* Fig. 4.1: fig04_01.c
   Counter-controlled repetition with the while statement */
#include <stdio.h>

int main()
{
   int counter = 1; // initialization

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

Program Output

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

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

int main()
{
   int counter; // define counter

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

Program Output

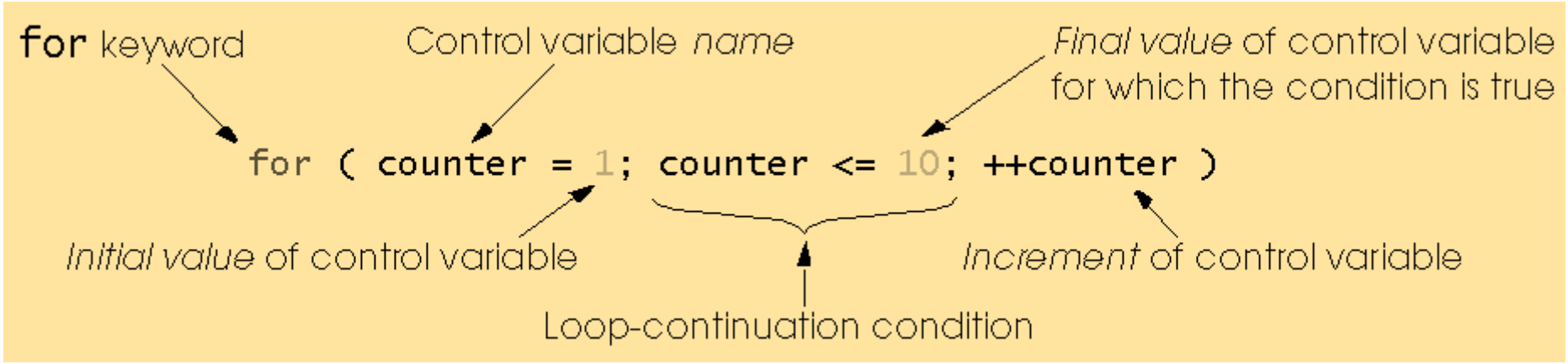
```
1
2
3
4
5
6
7
8
9
10
```

4.4 The for Repetition Statement

General Format of a `for` Statement

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

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



4.4 The `for` Repetition Statement

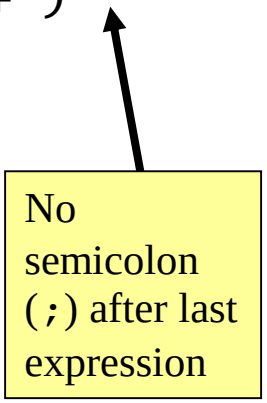
- Format when using `for` loops

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

- Example:

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

  – Prints the integers from one to ten



4.4 The `for` Repetition Statement

Comma-Separated Lists of Expressions

- Often, *expression1* and *expression3* are comma-separated lists of expressions.
- The commas as used here are actually 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.

- The comma operator is most often used in the `for` statement.
- 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.

Example: Comma-Separated Lists of Expressions in for

```
#include <stdio.h>

int main()
{
  int i, j;

  for ( (i=1, j=5) ;   (i<=10, j<=20) ; (i++, j=j+3) )
      printf("%d %d \n", i, j);
}
```

Program Output

```
1    5
2    8
3    11
4    14
5    17
6    20
```

4.4 The for Repetition Statement

Expressions in the `for` Statement's Header Are Optional

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

```
#include <stdio.h>
int main() {
  int i=1;
  for (  ;  i<=10 ;  )
  {
      printf("%d \n", i);
      i++;
  }
} // end main
```

Program Output

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

4.4 The `for` Repetition Statement

- For loops can usually be rewritten as while loops:

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

Increment Expression Acts Like a Standalone Statement

- The expression in the `for` statement acts like a stand-alone C statement at the end of the body of the `for`.
- Therefore, the expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

  are all equivalent in the increment part of the `for` statement.
- The two semicolons in the `for` statement are required.

4.4 The for Repetition Statement

- The body of the `for` statement could actually be merged into the rightmost portion of the `for` header by using the comma operator as follows:

```
for ( i=2; i <= 100; sum += i, i += 2 )
   ; // empty statement
```

- The initialization `sum = 0` could also be merged into the initialization section of the `for`.

```
#include <stdio.h>

int main() {
  int i, j;
  for ( i=1 ;  i<=10 ;  printf("%d \n", i) , i++ )
      ; // Empty statement
}
```

Program Output

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

4.5 The for Statement : 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 )
```

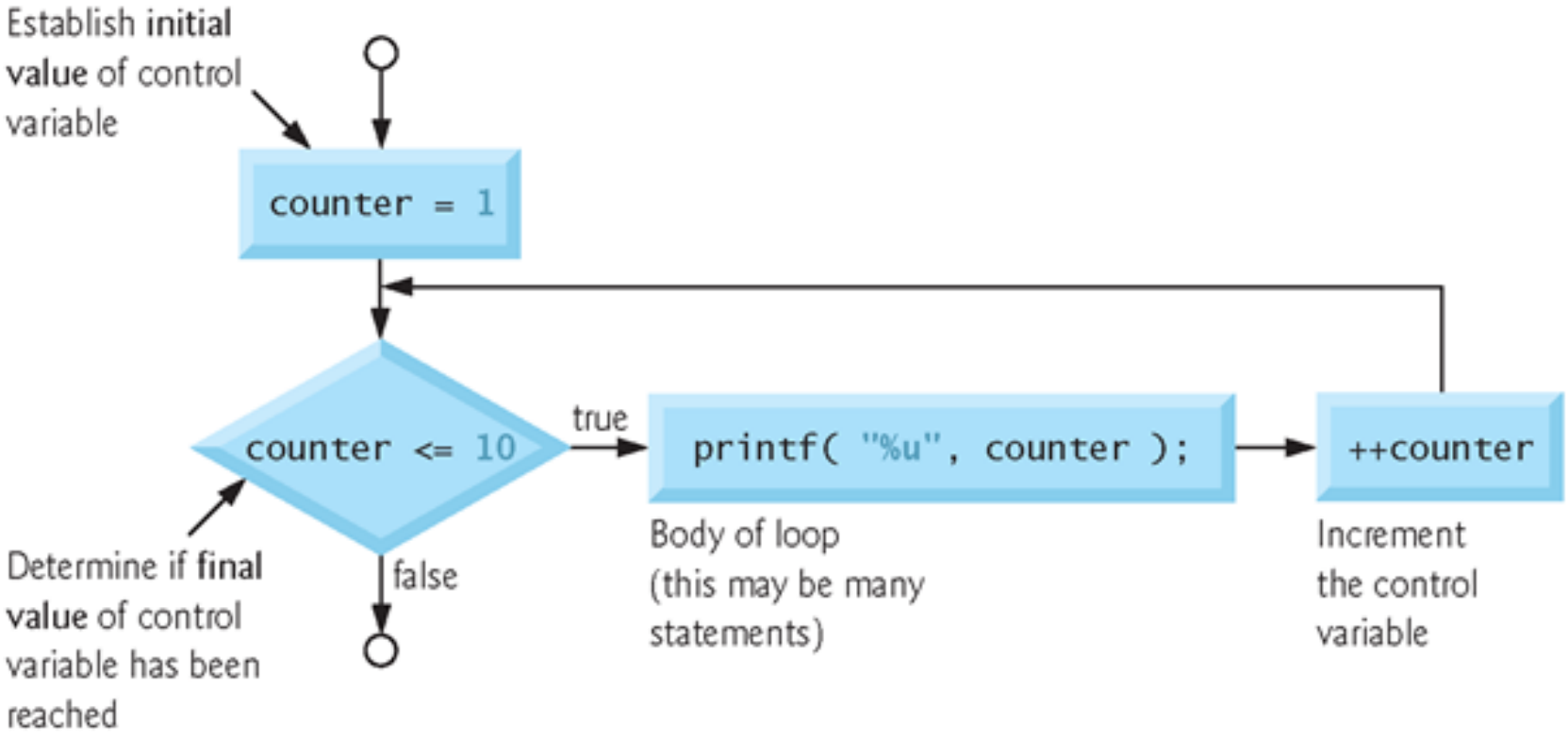
  is equivalent to

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

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

4.5 The for Statement : Notes and Observations

- The following is a typical flowcharting of the ***for*** repetition statement.



```c
/* Fig. 4.5: fig04_05.c
   Summation with for */
#include <stdio.h>
int main()
{
   int sum = 0; // initialize sum
   int number;  // number to be added to sum

   for ( number = 2; number <= 100; number += 2 ) {
      sum += number; // add number to sum
   }

   printf( "Sum is %d\n", sum ); // output sum
}
```

Program Output

```
Sum is 2550
```

Sum = 2+4+6+8+ . . . +100 = 2550

4.7 The switch Multiple-Selection Statement

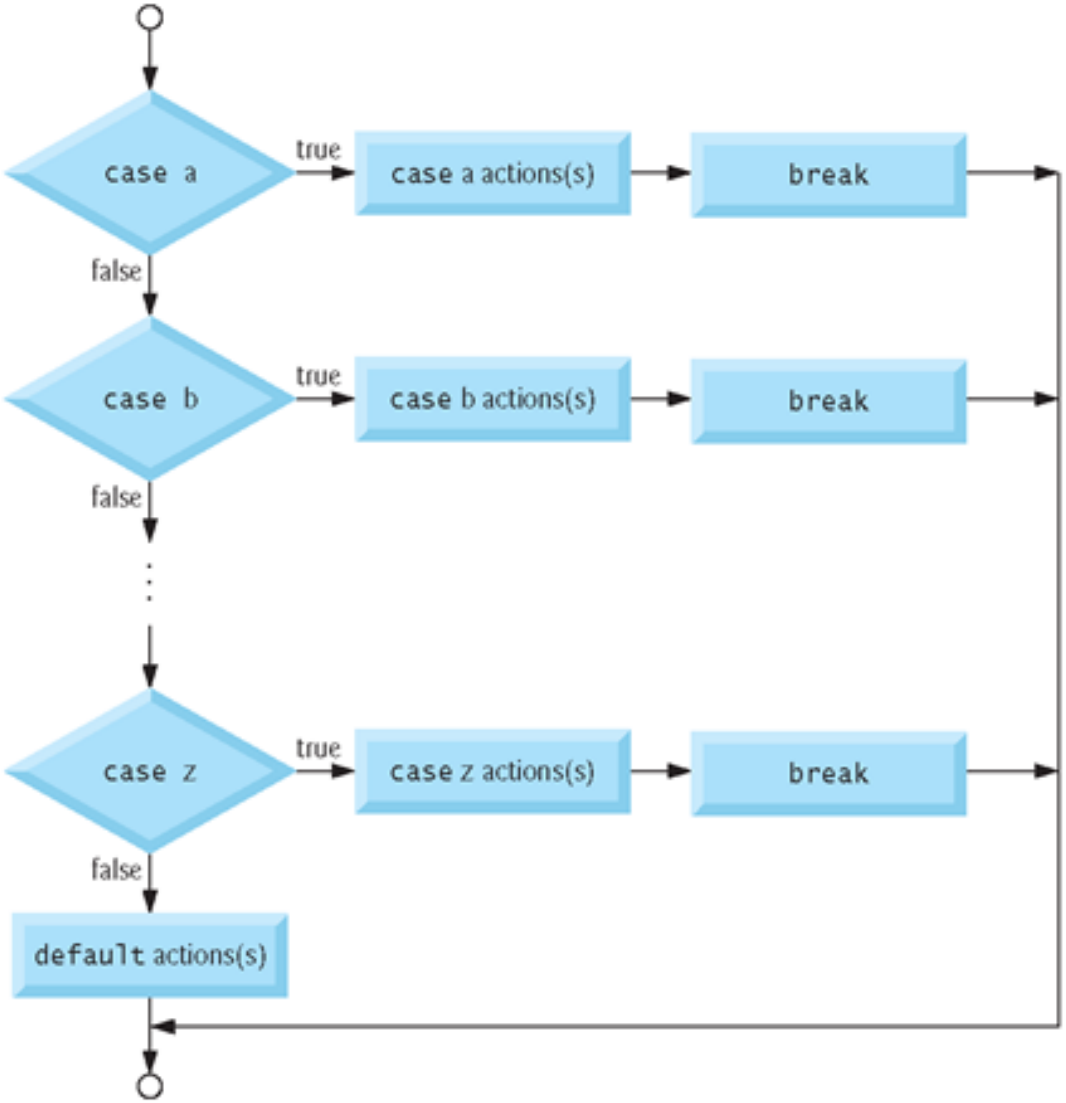
- switch
  - Useful when a variable or expression is tested for all the values it can assume and different actions are taken
- Format
  - Series of case labels and an optional default case

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

  - break; exits from statement

4.7 The switch Multiple-Selection Statement

- Flowchart of the `switch` statement



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

int main() {
   int grade;       // one grade
   int aCount = 0; // number of As
   int bCount = 0; // number of Bs
   int cCount = 0; // number of Cs
   int dCount = 0; // number of Ds
   int fCount = 0; // number of Fs

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

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

      // 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
```

Part 1 of 3

Sentinel input (EOF)

Control Z + ENTER

```
        case 'B':
        case 'b':
           ++bCount;
           break;

        case 'C':
        case 'c':
           ++cCount;
           break;

        case 'D':
        case 'd':
           ++dCount;
           break;

        case 'F':
        case 'f':
           ++fCount;
           break;

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

Part 2 of 3

Part 3 of 3

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

   } // end while

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

} // end main
```

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
^Z

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

4.8 The do…while Repetition Statement

- The do…while repetition statement
  - 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 Statement

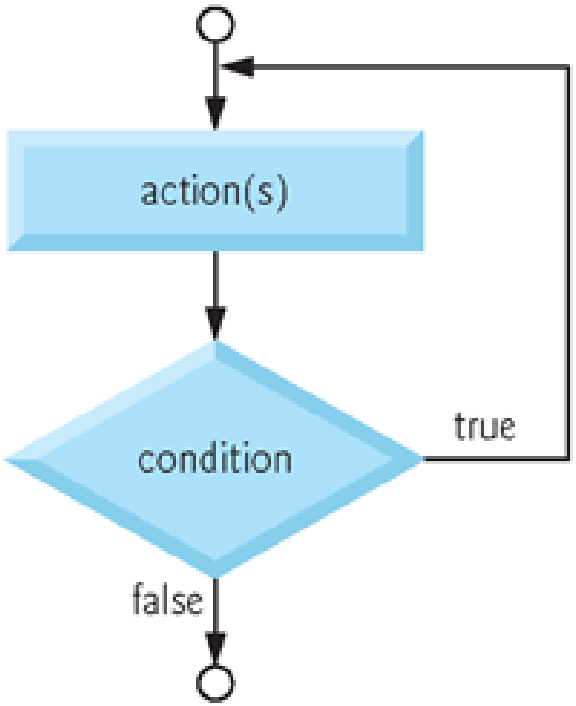
- Example (letting counter = 1):

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

  - Prints the integers from 1 to 10

4.8 The do…while Repetition Statement

- Flowchart of the do…while repetition statement



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

int main()
{
   int counter = 1;    // initialize counter

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

}
```

Program Output

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

4.9 The `break` and `continue` Statements

- `break`
  - Causes immediate exit from a `while`, `for`, `do…while` or `switch` statement
  - 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` statement

```c
/* Fig. 4.11: fig04_11.c
   Using the break statement in a for statement */
#include <stdio.h>
int main()
{
   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( "%d ", x ); // display value of x
   } // end for

   printf( "\nBroke out of loop at x == %d\n", x );
}
```

Program Output

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

4.9 The `break` and `continue` Statements

- `continue`
  - Skips the remaining statements in the body of a `while`, `for` or `do…while` statement
    - 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 statement */
#include <stdio.h>
int main()
{
   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

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

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

Program Output

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

The Unconditional Branch: goto

- Unstructured programming
  - Use when performance crucial
  - `break` to exit loop instead of waiting until condition becomes `false`

- `goto` statement
  - Changes flow control to first statement after specified label
  - A label is an identifier followed by a colon (i.e. `start:`)
  - Quick escape from deeply nested loop

```
goto start;
```

```c
/* Fig. 14.9: fig14_09.c
   Using goto */
#include <stdio.h>

int main()
{
   int count = 1; // initialize count

   start: // label

      if ( count > 10 ) {
         goto end;
      } // end if

      printf( "%d  ", count );
      count++;

      goto start; // goto the start line

   end: // label
   putchar( '\n' );
}
```

Notice how these are used

start:
end:
goto

Program Output

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

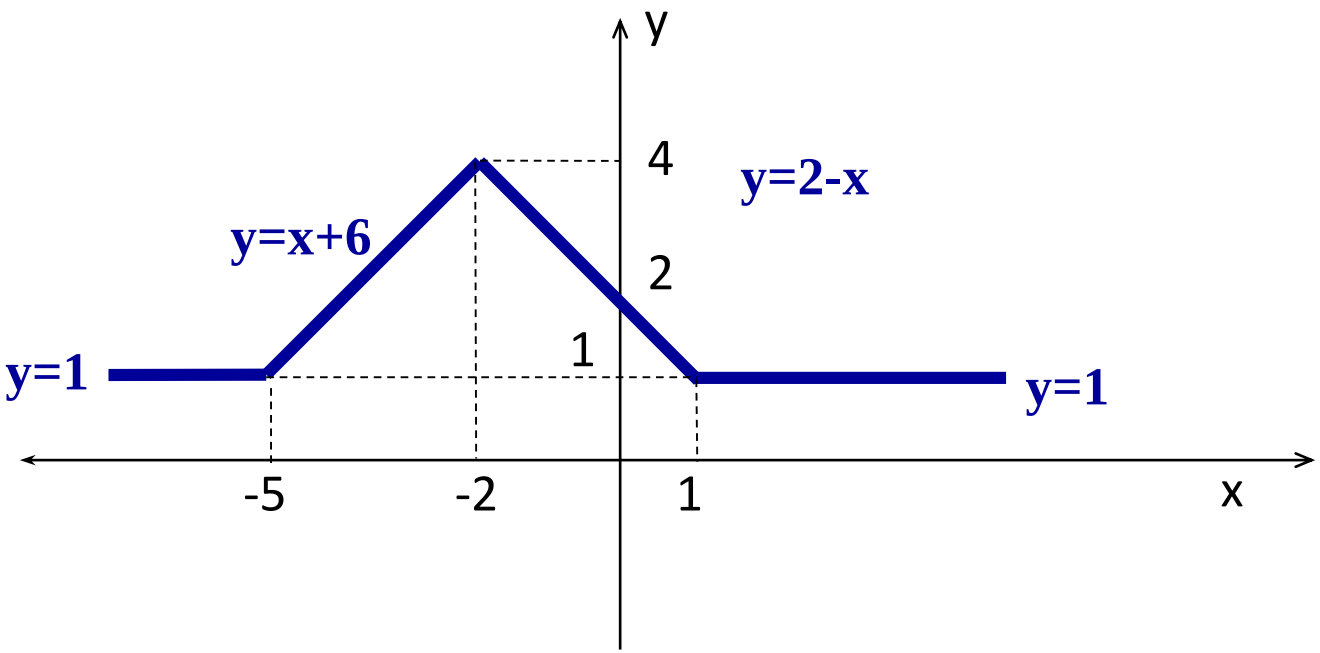
4.10 Logical Operators (Boolean)

- **&& (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`

Example

- Write a C program that produces the ***y=f(x)*** value regarding to the partial function shown below. Program should get an x value from user, and should print the corresponding y value.



```c
#include <stdio.h>

int main()
{
  int x, y ;

  printf("\n Enter x: ");
  scanf("%d", &x) ;

  if (x < -5 || x > 1)   y=1 ;

  if (x >= -5 && x < -2)   y = x+6 ;

  if (x >= -2 && x <= 1)   y = 2-x ;

  printf("y = %d \n", y);
}
```

Wrong syntax !

```c
if (-5 <= x < 2)
```

4.10 Logical Operators

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.

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.

expression	! expression
0	1
nonzero	0

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

4.10 Logical Operators

Operators						Associativity	Type
++	--	!	(type)			right to left	unary
*	/	%				left to right	multiplicative
+	-					left to right	additive
<	<=	>	>=			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.

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;
```