

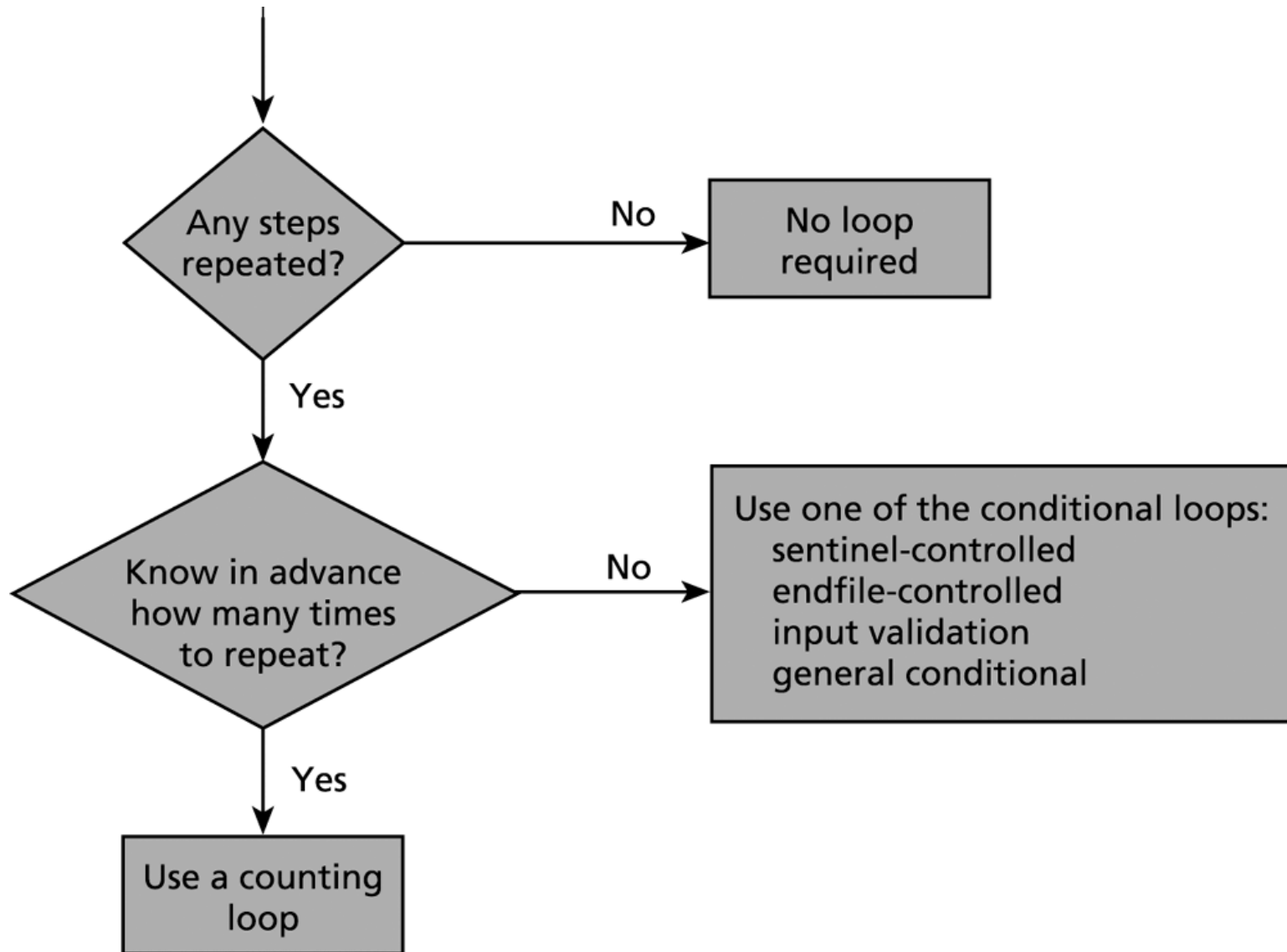
# Computer Programming

## Repetition and Loop Statements

- Controls the flow of program execution
  - Sequence
  - Selection
  - Repetition
- Repetition structure
  - Repetition of steps (loop body) : loop
  - while, for, and do-while statements
    - Each has advantages for some type of repetitions
  - Ex: calculate payroll for several employees

- How to design repetition
  - Solve the problem for a specific case
  - Try to generalize
  - Answer the following questions for repetition
    - Do I need to repeat any step?
    - How many times to repeat the steps?
    - How long to continue repetition?
  - Decide on the loop type based on the answers.
    - The flow chart on the next slide

# Loop Choice



# Comparison of Loop Kinds

**TABLE 5.1** Comparison of Loop Kinds

Kind	When Used	C Implementation Structures	Section Containing an Example
Counting loop	We can determine before loop execution exactly how many loop repetitions will be needed to solve the problem.	while for	5.2 5.4
Sentinel-controlled loop	Input of a list of data of any length ended by a special value	while, for	5.6
Endfile-controlled loop	Input of a single list of data of any length from a data file	while, for	5.6
Input validation loop	Repeated interactive input of a data value until a value within the valid range is entered	do-while	5.8
General conditional loop	Repeated processing of data until a desired condition is met	while, for	5.5, 5.9

# Counter Controlled Loop



- Repetition is managed by a loop control variable
  - For example a counter

General format:

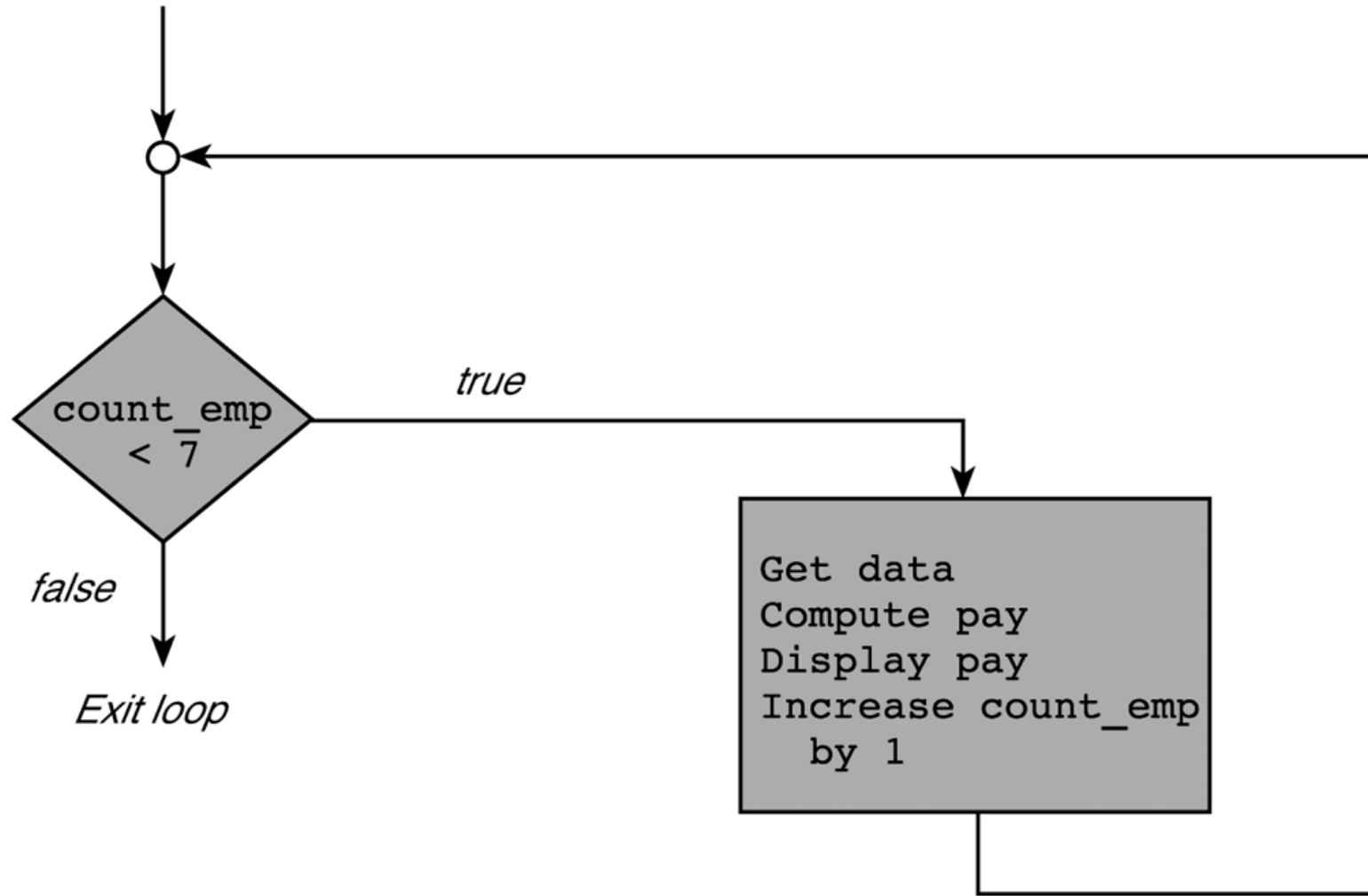
set counter to 0	//initialization
while counter < final value	//test
do something	//loop body
increase counter by one	//updating



# Program Fragment with a Loop

```
1. count_emp = 0;           /* no employees processed yet      */
2. while (count_emp < 7) {   /* test value of count_emp      */
3.     printf("Hours> ");
4.     scanf("%d", &hours);
5.     printf("Rate> ");
6.     scanf("%lf", &rate);
7.     pay = hours * rate;
8.     printf("Pay is $%6.2f\n", pay);
9.     count_emp = count_emp + 1; /* increment count_emp      */
10. }
11. printf("\nAll employees processed\n");
```

# Flowchart for a while Loop





# while statement



General syntax:

```
while (loop repetition control)
    statement
```

Example

```
count_star = 0;
while (count_star < N) {
    printf("*");
    count_star = count_star + 1;
}
```



- Calculate payroll for several employees
  - Calculate the total payroll as well
- Input:
  - For each employee
    - Hours, rate, pay
  - Number of employees
- Output
  - For each employee
    - Payroll
  - Total payroll

```

1.  /* Compute the payroll for a company */
2.
3.  #include <stdio.h>
4.
5.  int
6.  main(void)
7.  {
8.      double total_pay;      /* company payroll      */
9.      int     count_emp;     /* current employee */
10.     int     number_emp;    /* number of employees */
11.     double hours;          /* hours worked      */
12.     double rate;           /* hourly rate       */
13.     double pay;            /* pay for this period */
14.
15.     /* Get number of employees. */
16.     printf("Enter number of employees> ");
17.     scanf("%d", &number_emp);
18.
19.     /* Compute each employee's pay and add it to the payroll. */
20.     total_pay = 0.0;
21.     count_emp = 0;
22.     while (count_emp < number_emp) {
23.         printf("Hours> ");
24.         scanf("%lf", &hours);
25.         printf("Rate > $");
26.         scanf("%lf", &rate);
27.         pay = hours * rate;
28.         printf("Pay is $%6.2f\n\n", pay);
29.         total_pay = total_pay + pay;          /* Add next pay. */
30.         count_emp = count_emp + 1;
31.     }
32.     printf("All employees processed\n");
33.     printf("Total payroll is $%8.2f\n", total_pay);
34.
35.     return (0);
36. }

```

# Generalized conditional loop



- Ex: multiplying a list of numbers
  - Ask for numbers
  - Multiply as long as the product is less than 10000



# Generalized conditional loop



- Ex: multiplying a list of numbers
  - Ask for numbers
  - Multiply as long as the product is less than 10000

```
product = 1;
while (product < 10000){
    printf("%d \n Enter next item >", product);
    scanf("%d",&item);
    product = product * item;
}
```



# Compound assignment



## Simple assignment

```
count = count + 1;  
time = time - 1;  
product = product * item;  
n = n / (d + 1);  
value = value % 7;
```

In general:

`var = var op exp`

## Compound assignment

```
count += 1;  
time -= 1;  
product *= item;  
n /= (d + 1);  
value %= 7;
```

In general:

`var op= exp`



- for statement is another repetition structure
- supplies a designated space for each of the loop components
  - Initialization of the loop control variable
  - Test of the loop repetition control
  - Change of the loop control variable
- Syntax:  
    for (intialization expression;  
        loop repetition condition;  
        update expression)  
        statement;

# for Statement in a Counting Loop



```
1. /* Process payroll for all employees */
2. total_pay = 0.0;
3. for (count_emp = 0;                                /* initialization */
4.     count_emp < number_emp;                        /* loop repetition condition */
5.     count_emp += 1) {                             /* update */
6.     printf("Hours> ");
7.     scanf("%lf", &hours);
8.     printf("Rate > $");
9.     scanf("%lf", &rate);
10.    pay = hours * rate;
11.    printf("Pay is $%6.2f\n\n", pay);
12.    total_pay = total_pay + pay;
13. }
14. printf("All employees processed\n");
15. printf("Total payroll is $%8.2f\n", total_pay);
```





# for statement



```
for (count_star = 0;  
    count_star < N;  
    count_star += 1)  
    printf("*");
```

```
for (i = 0; i < max; i +=1)  
    printf("%d \n", i);
```

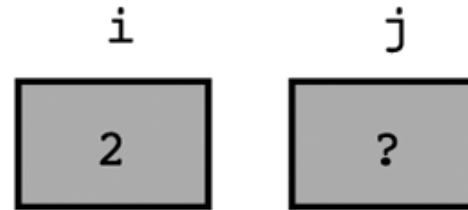
```
for (product = 1; product < 10000; product *= item)  
    scanf("%d", &item);
```



- Unary operators
- Side effect
  - ++ increments the operand
  - -- decrements the operand
- The value of the operation depends on the position of the operator
  - Pre-increment : operand is after the operator
    - Value is the variable's value after incrementing
  - Post-increment : operand is before the operator
    - Value is the variable's value before incrementing
  - Similar for decrement operator

# Prefix and Postfix Increments

Before...



Increments...

`j = ++i;`

prefix:  
Increment *i* and  
then use it.

`j = i++;`

postfix:  
Use *i* and then  
increment it.

After...



- What is the result of following code fragments

```
n = 4;  
printf("%3d", --n);  
printf("%3d", n);  
printf("%3d", n--);  
printf("%3d", n);  
y = n * 4 + ++n;  
x = n++ * --n;
```

- Write a function to compute factorial of an integer



# Function to Compute Factorial

```
1.  /*
2.  * Computes n!
3.  * Pre: n is greater than or equal to zero
4.  */
5.  int
6.  factorial(int n)
7.  {
8.      int i,          /* local variables */
9.      product;        /* accumulator for product computation */
10.
11.     product = 1;
12.     /* Computes the product n x (n-1) x (n-2) x ... x 2 x 1 */
13.     for (i = n; i > 1; --i) {
14.         product = product * i;
15.     }
16.
17.     /* Returns function result */
18.     return (product);
19. }
```

```

1.  /* Conversion of Celsius to Fahrenheit temperatures */
2.
3.  #include <stdio.h>
4.
5.  /* Constant macros */
6.  #define CBEGIN 10
7.  #define CLIMIT -5
8.  #define CSTEP 5
9.
10. int
11. main(void)
12. {
13.     /* Variable declarations */
14.     int    celsius;
15.     double fahrenheit;
16.
17.     /* Display the table heading */
18.     printf("    Celsius    Fahrenheit\n");
19.
20.     /* Display the table */
21.     ① for (celsius = CBEGIN;
22.     ②         celsius >= CLIMIT;
23.     ③         celsius -= CSTEP) {
24.     ④         fahrenheit = 1.8 * celsius + 32.0;
25.     ⑤         printf("%6c%3d%8c%7.2f\n", ' ', celsius, ' ', fahrenheit);
26.     }
27.
28.     return (0);
29. }

```

Celsius	Fahrenheit
10	50.00
5	41.00
0	32.00
-5	23.00

# Conditional Loops

- If you do not know exact number of repetitions
- Ex: ensuring valid user input
  - Continue to prompt user to enter a value as long as the response is not reasonable

Print an initial prompting message

Get the number of observed values

While the number of value is negative

Print a warning message and ask for another value

Get the number of observed values

- Where is initialization, test and update steps?
  - How to write the loop in C?

# Conditional Loops

- Ex: Monitoring gasoline supply
  - Capacity 80000 barrels
  - Use of gasoline is entered in gallons
    - 1 barrel = 42 gallons
  - Alert if the supply falls below 10% of the capacity
- Input:
  - Current supply
  - Several uses
- Output
  - Remaining supply
  - Alert



**TABLE 5.5** Problem-Solving Questions for Loop Design

Question	Answer	Implications for the Algorithm
What are the inputs?	Initial supply of gasoline (barrels). Amounts removed (gallons).	Input variables needed: <code>start_supply</code> <code>remov_gals</code> Value of <code>start_supply</code> must be input once, but amounts removed are entered many times.
What are the outputs?	Amounts removed in gallons and barrels, and the current supply of gasoline.	Values of <code>current</code> and <code>remov_gals</code> are echoed in the output. Output variable needed: <code>remov_brls</code>
Is there any repetition?	Yes. One repeatedly 1. gets amount removed 2. converts the amount to barrels 3. subtracts the amount removed from the current supply 4. checks to see whether the supply has fallen below the minimum.	Program variable needed: <code>min_supply</code>
Do I know in advance how many times steps will be repeated?	No.	Loop will not be controlled by a counter.
How do I know how long to keep repeating the steps?	As long as the current supply is not below the minimum.	The loop repetition condition is <code>current &gt;= min_supply</code>

```

1.  /*
2.   * Monitor gasoline supply in storage tank. Issue warning when supply
3.   * falls below MIN_PCT % of tank capacity.
4.   */
5.
6.  #include <stdio.h>
7.
8.  /* constant macros */
9.  #define CAPACITY      80000.0 /* number of barrels tank can hold      */
10. #define MIN_PCT       10      /* warn when supply falls below this
11.                               percent of capacity                      */
12. #define GALS_PER_BRL  42.0    /* number of U.S. gallons in one barrel */
13.
14. /* Function prototype */
15. double monitor_gas(double min_supply, double start_supply);
16.
17. int
18. main(void)
19. {
20.     double start_supply, /* input - initial supply in barrels      */
21.            min_supply,   /* minimum number of barrels left without
22.                               warning                      */
23.            current;      /* output - current supply in barrels      */
24.
25.     /* Compute minimum supply without warning */
26.     min_supply = MIN_PCT / 100.0 * CAPACITY;
27.
28.     /* Get initial supply */
29.     printf("Number of barrels currently in tank> ");
30.     scanf("%lf", &start_supply);
31.
32.     /* Subtract amounts removed and display amount remaining
33.        as long as minimum supply remains.
34.     */
35.     current = monitor_gas(min_supply, start_supply);
36.
37.     /* Issue warning
38.     */
39.     printf("only %.2f barrels are left.\n\n", current);
40.     printf("*** WARNING ***\n");

```

```

39.     printf("Available supply is less than %d percent of tank's\n",
40.           MIN_PCT);
41.     printf("%.2f-barrel capacity.\n", CAPACITY);
42.
43.     return (0);
44. }
45.
46. /*
47.  * Computes and displays amount of gas remaining after each delivery
48.  * Pre : min_supply and start_supply are defined.
49.  * Post: Returns the supply available (in barrels) after all permitted
50.  *       removals. The value returned is the first supply amount that is
51.  *       less than min_supply.
52.  */
53. double
54. monitor_gas(double min_supply, double start_supply)
55. {
56.     double remov_gals,    /* input - amount of current delivery      */
57.           remov_brls,    /*           in barrels and gallons */
58.           current;       /* output - current supply in barrels */
59.
60.     for (current = start_supply;
61.          current >= min_supply;
62.          current -= remov_brls) {
63.         printf("%.2f barrels are available.\n\n", current);
64.         printf("Enter number of gallons removed> ");
65.         scanf("%lf", &remov_gals);
66.         remov_brls = remov_gals / GALS_PER_BRL;
67.
68.         printf("After removal of %.2f gallons (%.2f barrels),\n",
69.               remov_gals, remov_brls);
70.     }
71.
72.     return (current);
73. }

```

# Sentinel Controlled Loops



- Input one additional data item at each repetition
  - Usually number of items is not known in advance
  - When to stop reading data?
- Sentinel value: unique value to stop repetition
  - Should be an abnormal value

Get a line of data

While the sentinel value has not been encountered

Process the data line

Get another line of data

- Where is initialization, test and update stages



# Sentinel Controlled Loops



- Ex: Calculate sum of a collection of exam scores
  - Assume the number of students is not known
  - What is the sentinel value?
- Input:
  - Exam score
- Output:
  - Sum of scores



# Sentinel Controlled Loops



Algorithm:

Initialize sum to zero

while score is not the sentinel

    Get score

    Add score to sum



# Sentinel Controlled Loops



Correct Algorithm:

Initialize sum to zero

Get the first score

while score is not the sentinel

    Add score to sum

    Get score



# Sentinel-Controlled while Loop



```
1.  /* Compute the sum of a list of exam scores. */
2.
3.  #include <stdio.h>
4.
5.  #define SENTINEL -99
6.
7.  int
8.  main(void)
9.  {
10.     int sum = 0,    /* output - sum of scores input so far          */
11.        score;       /* input - current score          */
12.
13.     /* Accumulate sum of all scores.                                */
14.     printf("Enter first score (or %d to quit)> ", SENTINEL);
15.     scanf("%d", &score);      /* Get first score.          */
16.     while (score != SENTINEL) {
17.         sum += score;
18.         printf("Enter next score (%d to quit)> ", SENTINEL);
19.         scanf("%d", &score);  /* Get next score.          */
20.     }
21.     printf("\nSum of exam scores is %d\n", sum);
22.
23.     return (0);
24. }
```





# Sentinel-Controlled for Loop



- Can we use for statement for sentinel controlled loops?

```
/* Accumulate sum of all scores. */
printf("Enter first score (or %d to quit)> ", SENTINEL);
scanf("%d", &score);          /* Get first score. */
while (score != SENTINEL) {
    sum += score;
    printf("Enter next score (%d to quit)> ", SENTINEL);
    scanf("%d", &score);      /* Get next score. */
}
printf("\nSum of exam scores is %d\n", sum);
```



# Sentinel-Controlled for Loop



```
/* Accumulate sum of all scores.                                     */
printf("Enter first score (or %d to quit)> ", SENTINEL);
scanf("%d", &score);          /* Get first score.                  */
while (score != SENTINEL) {
    sum += score;
    printf("Enter next score (%d to quit)> ", SENTINEL);
    scanf("%d", &score);      /* Get next score.          */
}
printf("\nSum of exam scores is %d\n", sum);
```

```
printf(.....);
for (scanf("%d",&score);
     score != SENTINEL;
     scanf("%d",&score)) {
    sum += score;
    printf(.....);
}
```



# End-file Controlled Loops

Ex: Calculate sum of a list of integers in a file

- A data file is terminated by an endfile character
  - detected by fscanf functions.
- special sentinel value is not required
  - uses the status value returned by fscanf

Algorithm:

Initialize sum to zero

Read the first value

while end of file is not reached

    Add value to sum

    Read the next value

# End-file Controlled Loops



```
1.  /*
2.   *   Compute the sum of the list of exam scores stored in the
3.   *   file scores.dat
4.   */
5.
6.  #include <stdio.h>          /* defines fopen, fclose, fscanf,
7.                               fprintf, and EOF
8.
9.  int
10. main(void)
11. {
12.     FILE *inp;               /* input file pointer
13.     int    sum = 0,          /* sum of scores input so far
14.           score,             /* current score
15.           input_status;      /* status value returned by fscanf
16.     inp = fopen("scores.dat", "r");
17.
18.     printf("Scores\n");
19.
20.     input_status = fscanf(inp, "%d", &score);
21.     while (input_status != EOF) {
22.         printf("%5d\n", score);
23.         sum += score;
24.         input_status = fscanf(inp, "%d", &score);
25.     }
26.
27.     printf("\nSum of exam scores is %d\n", sum);
28.     fclose(inp);
29.
30.     return (0);
31. }
```



# Infinite Loop on Faulty Data



- If the file contains a faulty data 7o, fscanf
  - stops at the letter 'o',
  - stores the value 7 in score
  - leaves the letter 'o' unprocessed.
  - returns a status value of one
- On the next loop iteration, fscanf
  - finds the letter 'o' awaiting processing
  - leaves the variable score unchanged
  - leaves the letter ' o ' unprocessed,
  - returns a status value of zero
- In the previous program
  - the return value of fscanf is not checked for values other than EOF
  - unsuccessful attempt to process the letter 'o' repeats over and over.

Infinite loop!...



# Infinite Loop on Faulty Data



- Solution: Change the loop repetition condition to  
    while (input\_status == 1)
- loop exits on
  - end of file (input\_status negative) OR
  - faulty data (input\_status zero)
- Add an if statement after the loop to decide whether to print the results or to warn of bad input.

```
if (input_status == EOF)
    printf ('Sum of exam scores is %d\n", sum);
else {
    fscanf (inp, "%c", &bad_char);
    printf("*** Error in input: %c ***\n", bad_char);
```



# Nested Loops



- Loops may be nested like other control structures.
  - an outer loop with one or more inner loops.
  - Each time the outer loop is repeated, the inner loops are reentered,
- Ex: Audubon Club members' sightings of bald eagles
- Input: for each month a group of integers followed by a zero
- Output: for each month total sightings
- program contains a sentinel loop (for sightings in a month) nested within a counting loop (for months).



# Nested Loops



```
5.
6. #include <stdio.h>
7.
8. #define SENTINEL    0
9. #define NUM_MONTHS 12
10.
11. int
12. main(void)
13. {
14.
15.     int month,          /* number of month being processed          */
16.         mem_sight,      /* one member's sightings for this month      */
17.         sightings;      /* total sightings so far for this month      */
18.
19.     printf("BALD EAGLE SIGHTINGS\n");
20.     for (month = 1;
21.          month <= NUM_MONTHS;
22.          ++month) {
23.         sightings = 0;
24.         scanf("%d", &mem_sight);
25.         while (mem_sight != SENTINEL) {
26.             if (mem_sight >= 0)
27.                 sightings += mem_sight;
28.             else
29.                 printf("Warning, negative count %d ignored\n",
30.                        mem_sight);
31.             scanf("%d", &mem_sight);
32.         } /* inner while */
33.
34.         printf("  month %2d: %2d\n", month, sightings);
35.     } /* outer for */
36.
37.     return (0);
38. }
```



# Nested Loops

- Ex: a simple program with two nested counting loops.
  - The outer loop is repeated three times (for  $i = 1, 2, 3$ ).
  - The number of times the inner loop is repeated depends on the current value of  $i$ .

# Nested Loops

```
1.  /*
2.   * Illustrates a pair of nested counting loops
3.   */
4.
5.  #include <stdio.h>
6.
7.  int
8.  main(void)
9.  {
10.     int i, j;    /* loop control variables */
11.
12.     printf("          I      J\n");          /* prints column labels          */
13.
14.     for (i = 1; i < 4; ++i) {                  /* heading of outer for loop          */
15.         printf("Outer %6d\n", i);
16.         for (j = 0; j < i; ++j) {              /* heading of inner loop              */
17.             printf("  Inner%9d\n", j);
18.         } /* end of inner loop */
19.     } /* end of outer loop */
20.
21.     return (0);
22. }
```

# Nested Loops

- The output of the algorithm:

	I	J
Outer	1	
Inner		0
Outer	2	
Inner		0
Inner		1
Outer	3	
Inner		0
Inner		1
Inner		2

---

# Nested Loops

- What is displayed by the following program segments, assuming m is 3 and n is 5?

```
a.  for (i = 1; i <= n; ++i) {  
      for (j = 0; j < i; ++j) {  
          printf("*");  
      }  
      printf("\n");  
  }  
b.  for (i = n; i > 0; --i) {  
      for (j = m; j > 0; --j) {  
          printf("*");  
      }  
      printf("\n");  
  }
```

- for statements and while statements evaluate loop repetition condition before the first execution of the loop body.
- Pretest is usually desirable
  - when there may be no data items to process
  - when the initial value of the loop control variable is outside its expected range.
- Sometimes loop must execute at least once
- Ex: interactive input
  1. Get a data value.
  2. If data value isn't in the acceptable range, go back to step 1.

- C provides the do-while statement to implement such loops
  1. Get a data value.
  2. If data value isn't in the acceptable range, go back to step 1.

```
do {  
    printf("Enter a letter from A to E> ");  
    scanf("%c", &letter);  
} while (letter < 'A' || letter > 'E');
```

- SYNTAX:

```
do {  
    statements  
} while ( loop repetition condition );
```

- Ex: Find first even input

```
do  
    status = scanf("%d", &num);  
while (status > 0 && (num % 2) != 0);
```

- If loop repetition condition is complex
  - Use a flag is a type int (values: **1** (true) and **0** (false))
  - Flag represents whether a certain event has occurred.
  
- Ex: Input Validation
  - The do-while is often used in checking for valid input
    - An input is always needed
  - Two nested loops
    - Repeat reading input when the input is not valid
      - not in range OR not a number
    - Repeat reading input to skip invalid input line
      - Not to have infinite loop



```

6.  int
7.  get_int (int n_min, int n_max)
8.  {
9.      int  in_val,                /* input - number entered by user      */
10.         status;                 /* status value returned by scanf     */
11.      char skip_ch;               /* character to skip                   */
12.      int  error;                 /* error flag for bad input           */
13.      /* Get data from user until in_val is in the range. */
14.      do {
15.          /* No errors detected yet. */
16.          error = 0;
17.          /* Get a number from the user. */
18.          printf("Enter an integer in the range from %d ", n_min);
19.          printf("to %d inclusive> ", n_max);
20.          status = scanf("%d", &in_val);
21.
22.          /* Validate the number. */
23.          if (status != 1) { /* in_val didn't get a number */
24.              error = 1;
25.              scanf("%c", &skip_ch);
26.              printf("Invalid character >>%c>>. ", skip_ch);
27.              printf("Skipping rest of line.\n");
28.          } else if (in_val < n_min || in_val > n_max) {
29.              error = 1;
30.              printf("Number %d is not in range.\n", in_val);
31.          }
32.
33.          /* Skip rest of data line. */
34.          do
35.              scanf("%c", &skip_ch);
36.          while (skip_ch != '\n');
37.      } while (error);
38.
39.      return (in_val);
40. }

```

- Do they behave similarly? Why?

```
scanf("%d", &num);  
while (num != SENT) {  
    /* process num */  
    scanf("%d", &num);  
}
```

```
do {  
    scanf("%d", &num);  
    /* process num */  
} while (num != SENT);
```

- Which of the following code is better way to implement a sentinel-controlled loop? Why?

```
scanf("%d", &num);  
while (num != SENT) {  
    /* process num */  
    scanf("%d", &num);  
}
```

```
do {  
    scanf("%d", &num);  
    if (num != SENT)  
        /* process num */  
} while (num != SENT);
```

- Rewrite the following code using do-while statement with no decisions in the loop body:

```
sum = 0;  
for (odd = 1; odd < n; odd+=2)  
    sum += odd;
```

## Problem: Collecting area for Solar-Heated House

- Area depends on several factors
  - the average number of heating degree days for each month
    - the product of the average difference between inside and outside temperatures and the number of days in the month
  - the average solar insolation for each month
    - rate at which solar radiation falls on one square foot of a given location
  - heating requirement per square foot of floor space
  - floor space
  - efficiency of the collection method

# Case Study:

- The formula for the desired collecting area (A)  
$$A = \text{heat loss} / \text{energy source}$$
- *heat lost* is the product of the heating requirement, the floor space, and the heating degree days.
- *energy resource* is the product of the efficiency of the collection method, the average solar insolation per day and the number of days.
- Two data files
  - hdd.txt contains numbers representing the *average* heating degree days for each months.
  - solar.txt contains the average solar insolation for each month

# Case Study:



## Problem Inputs

Average heating degree days file

Average solar insolation file

heat\_deg\_days /\* average heating degree days for coldest month \*/

coldest\_mon /\* coldest month (number 1..12)

solar\_insol /\* average daily solar insolation for coldest month\*/

heating\_req /\* Btu/degree day Ft<sup>2</sup> \*/

efficiency /\* % of solar insolation converted to usable heat \*/

floor\_space /\* square feet \*/

## Program Variables

energy\_resrc /\* usable solar energy available in coldest month  
(Btus obtained from 1 Ft<sup>2</sup> of collecting area) \*/

## Problem Outputs

heat\_loss /\* Btus of heat lost by structure in coldest month \*/

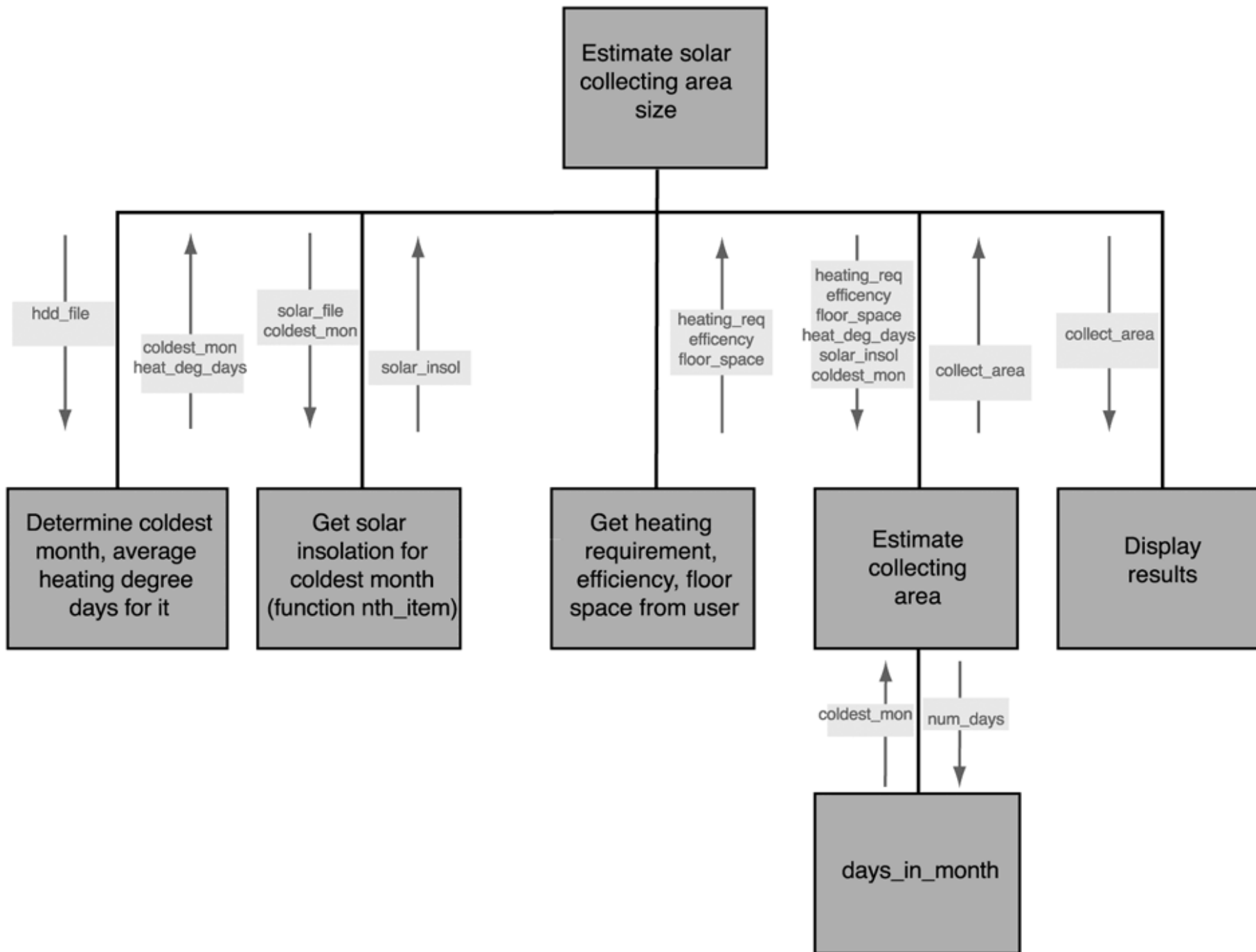
collect\_area /\* approximate size Ft<sup>2</sup> of collecting area needed \*/



## ■ Algorithm

1. Determine the coldest month and the average heating degree days for this month.
2. Find the average daily solar insolation per  $\text{Ft}^2$  for the coldest month.
3. Get from the user the other problem inputs:  
heating\_req, efficiency, floor\_space.
  1. Estimate the collecting area needed.
  2. Display results.





# Case Study:



## ■ Program to Approximate Solar Collecting Area Size

```
1.  /*
2.   * Estimate necessary solar collecting area size for a particular type of
3.   * construction in a given location.
4.   */
5.  #include <stdio.h>
6.
7.  int days_in_month(int);
8.  int nth_item(FILE *, int);
9.
```

*(continued)*



```

10. int main(void)
11. {
12.     int heat_deg_days, /* average for coldest month */
13.         solar_insol,   /* average daily solar radiation per
14.                        ft^2 for coldest month */
15.         coldest_mon,   /* coldest month: number in range 1..12 */
16.         heating_req,   /* Btu / degree day ft^2 requirement for
17.                        given type of construction */
18.         efficiency,    /* % of solar insolation converted to
19.                        usable heat */
20.         collect_area,  /* ft^2 needed to provide heat for
21.                        coldest month */
22.         ct,            /* position in file */
23.         status,        /* file status variable */
24.         next_hdd;      /* one heating degree days value */
25.     double floor_space, /* ft^2 */
26.         heat_loss,      /* Btus lost in coldest month */
27.         energy_resrc;   /* Btus heat obtained from 1 ft^2
28.                        collecting area in coldest month */
29.     FILE *hdd_file;     /* average heating degree days for each
30.                        of 12 months */
31.     FILE *solar_file;   /* average solar insolation for each of
32.                        12 months */
33.
34.     /* Get average heating degree days for coldest month from file */
35.     hdd_file = fopen("hdd.txt", "r");
36.     fscanf(hdd_file, "%d", &heat_deg_days);
37.     coldest_mon = 1;
38.     ct = 2;
39.     status = fscanf(hdd_file, "%d", &next_hdd);
40.     while (status == 1) {
41.         if (next_hdd > heat_deg_days) {
42.             heat_deg_days = next_hdd;
43.             coldest_mon = ct;
44.         }
45.
46.         ++ct;
47.         status = fscanf(hdd_file, "%d", &next_hdd);
48.     }
49.     fclose(hdd_file);
50.

```

```

51.  /* Get corresponding average daily solar insolation from other file */
52.  solar_file = fopen("solar.txt", "r");
53.  solar_insol = nth_item(solar_file, coldest_mon);
54.  fclose(solar_file);
55.
56.  /* Get from user specifics of this house */
57.  printf("What is the approximate heating requirement (Btu / ");
58.  printf("degree day ft^2)\nof this type of construction?\n=> ");
59.  scanf("%d", &heating_req);
60.  printf("What percent of solar insolation will be converted ");
61.  printf("to usable heat?\n=> ");
62.  scanf("%d", &efficiency);
63.  printf("What is the floor space (ft^2)?\n=> ");
64.  scanf("%lf", &floor_space);
65.
66.  /* Project collecting area needed */
67.  heat_loss = heating_req * floor_space * heat_deg_days;
68.  energy_resrc = efficiency * 0.01 * solar_insol *
69.    days_in_month(coldest_mon);
70.  collect_area = (int)(heat_loss / energy_resrc + 0.5);
71.
72.  /* Display results */
73.  printf("To replace heat loss of %.0f Btu in the ", heat_loss);
74.  printf("coldest month (month %d)\nwith available ", coldest_mon);
75.  printf("solar insolation of %d Btu / ft^2 / day,", solar_insol);
76.  printf(" and an\nefficiency of %d percent,", efficiency);
77.  printf(" use a solar collecting area of %d", collect_area);
78.  printf(" ft^2.\n");
79.
80.  return 0;
81. }
82.

```

```

83.  /*
84.   * Given a month number (1 = January, 2 = February, ...,
85.   * 12 = December ), return the number of days in the month
86.   * (nonleap year).
87.   * Pre: 1 <= monthNumber <= 12
88.   */
89. int days_in_month( int month_number )
90. {
91.
92.     int ans;
93.
94.     switch (month_number) {
95.     case 2: ans = 28; /* February */
96.             break;
97.
98.     case 4: /* April */
99.     case 6: /* June */
100.    case 9: /* September */
101.    case 11: ans = 30; /* November */
102.            break;
103.
104.    default: ans = 31;
105.    }
106.
107.    return ans;
108. }
109.
110. /*
111.  * Finds and returns the nth integer in a file.
112.  * Pre: data_file accesses a file of at least n integers (n >= 1).
113.  */
114. int nth_item(FILE *data_file, int n)
115. {
116.     int i, item;
117.
118.     for (i = 1; i <= n; ++i)
119.         fscanf(data_file, "%d", &item);
120.
121.     return item;
122. }

```

- Error Types:
  - syntax errors
  - run time errors
  - logic errors
- run-time error or logic error is usually not obvious
  - you may spend considerable time and energy locating it.
- Method:
  - examine the program output and determine program part generating incorrect results
  - focus on the statements and try to determine the fault
- OR
  - Use Debugger programs
  - Debug without debugger

## Of-by-one Loop Errors

- A common logic error with loops
  - loop executes one more time or one less time than required
- In sentinel-controlled loops, an extra repetition is more dangerous.
- In counting loops, the initial and final values of counter should be correct and the loop repetition condition should be right.
- Ex: the following loop body executes  $n + 1$  times instead of  $n$  times.

```
for (i=0; i <= n; ++i)
    sum += i;
```

- Don't Confuse
  - Use if statement to implement decision step !!
  - Use while statement to implement loop !!
- In using while or for statements, don't forget that
  - The structure assumes that the loop body is a single statement!!
  - Use (always) braces for consisting multiple statements !!
- Keep in mind that compiler ignore indentation!!
  - Ex : x is 1000 and max is 0;

Wrong!! (infinite loop)

```
while (x > max)
    sum+=x;
    x++;
```

True

```
while (x > max) {
    sum+=x;
    x++;
}
```



# Common Programming Errors



- Don't forget!!

= : is assignment operator

== : is equality operator

- Wrong!!

True

while (x=1)

while (x==1)

.....

.....

