Introduction to Computers & Programming

Comp 1330/ First Semester 2024/2025

Instructor: Saif Harbia

Faculty of Engineering and Technology
Department of Computer Science

Chapter 05

Repetition and Loop Statements

Chapter Objectives:

1. Learn about **repetition** as an important control structure in programming.
2. Loop **control variables** and the **three steps** needed to control loop repetition.
3. To learn how to use the C **for , while** , and **do-while**.
4. Learn common loop patterns such as **counting loops, sentinel-controlled loops,** and **flag-controlled loops**.
5. How to **debug** programs using a debugger and diagnostic output statement.

REPETITION

- In your programs so far, the statements in the program body execute only once. However, in most commercial software that you use, you can repeat a process many times.

- **Repetition**, you'll recall, is the third type of program control structure (***sequence , selection , repetition***).

- The repetition of steps in a program is called a **loop .**

- We describe three **C loop control statements**: **while** , **for** , and **do – while**.

5.1 REPETITION IN PROGRAMS

➢ Ask yourself some of the following questions to determine whether loops will be required in the general algorithm: (1)

1. Were there any steps I repeated as I solved the problem? If so, which ones?

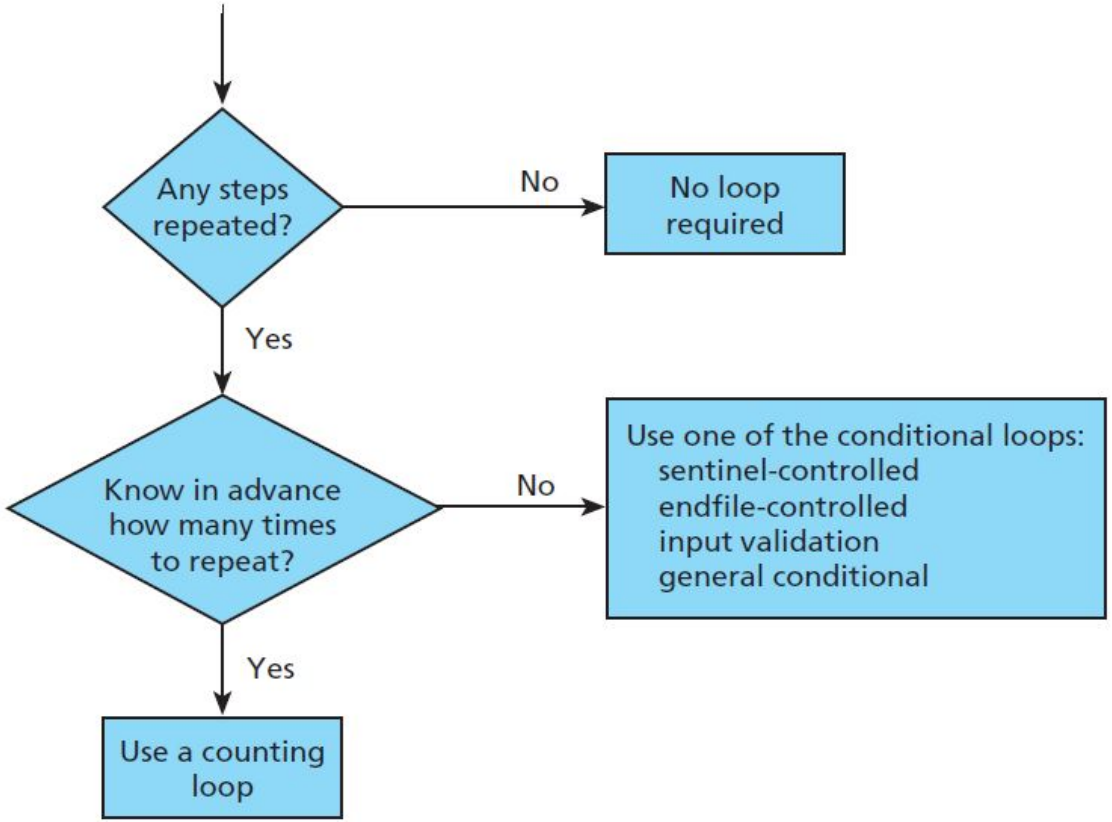
2. If the answer to question 1 is yes, did I know in advance how many times to repeat the steps?

3. If the answer to question 2 is no, how did I know how long to keep repeating the steps?

➢ See Table 5.1 p.237 for Comparison of Loop Kinds

FIGURE 5.1

Flow Diagram of Loop Choice Process



5.2 COUNTING LOOPS AND THE WHILE STATEMENT

- **Counter-controlled loop** (**or counting loop**) because its repetition is managed by **a loop control variable** whose value represents a **count**.

- Follows this general format:

Set *loop control variable* to an initial value of 0 .
while *loop control variable* < *final value*
    Do Something. . .
    Increase *loop control variable* by 1 .

THE WHILE STATEMENT

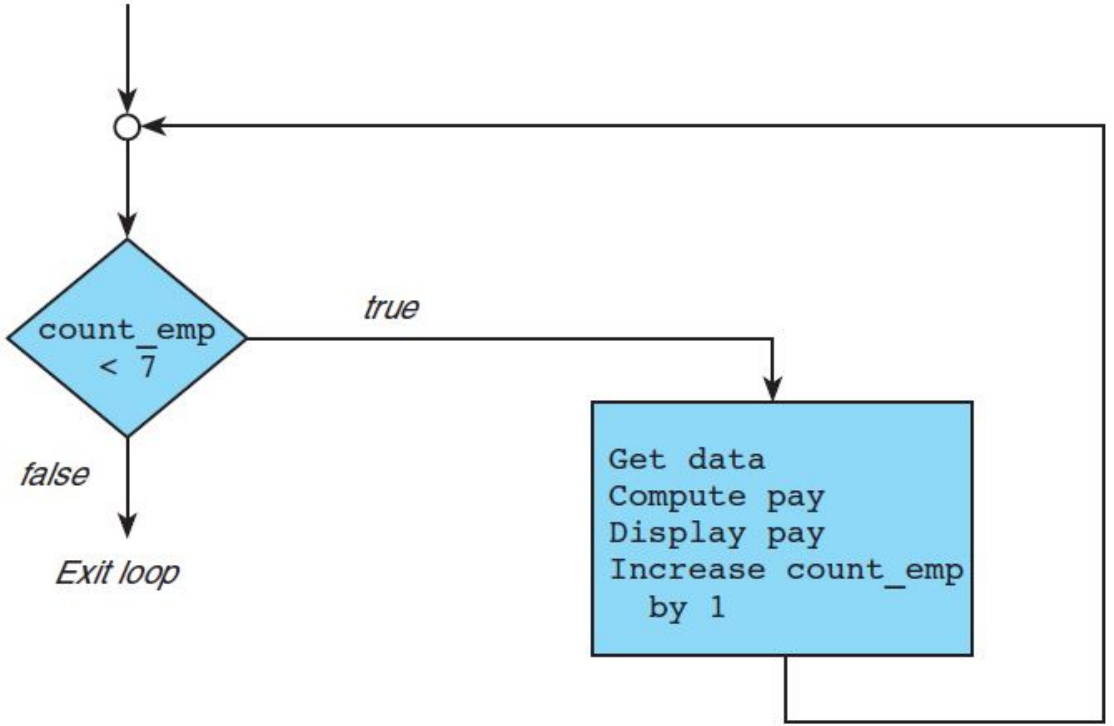
FIGURE 5.2 Program Fragment With a Loop

```
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");
```

The expression following the reserved word while is called the **loop repetition condition .** (1)

FIGURE 5.3

Flowchart for a while Loop




```
if (count_emp < 7) {      /* what is the difference */ (1)
. . .
}
```

SYNTAX OF THE WHILE STATEMENT

- In Fig. 5.2 the variable **count_emp** is called the **loop control variable** (1)
- The loop control variable **count_emp** must be **(1) initialized, (2) tested, and (3) updated** for the loop to execute properly.

- **Initialization**. **count_emp** is set to an initial value of 0 (initialized to 0) before the while statement is reached.

- **Testing**. **count_emp** is tested before the start of each loop repetition (called an iteration or a pass).

- **Updating**. **count_emp** is updated (its value increased by 1) during each iteration.

- If the loop control variable is not updated, the loop will execute "**forever**." Such a loop is called an **infinite loop**

➢ **Example:**

```
/* Display n asterisks. */
Printf("Enter value for n");
Scanf("%d", &n));
count_star = 0;
while (count_star < n) {
   printf("*");
   count_star = count_star + 1;
}
```

Note: If *loop repetition condition evaluates to false the first time it is tested, statement is not executed*.

5.3 COMPUTING A SUM OR A PRODUCT IN A LOOP

- **Example 5.1 p.242: See FIGURE 5.4 p.242 for the program.**

➢ **total_pay** is an **accumulator variable**, and it accumulates the total payroll value.

➢ Initializing **total_pay** to 0 is critical.(1)

➢ **total_pay = total_pay + pay;** /* Add next pay. */
adds the current value of pay to the sum being accumulated in **total_pay** .

- This loop is more general than the one in Figure 5.2.(2)

MULTIPLYING A LIST OF NUMBERS

- **Example 5.2 p.245:**

➢ **This loop is an example of the general conditional loop, whose pseudocode is shown below.**

1. **Initialize loop control variable.**
2. **As long as exit condition hasn't been met**
3. **Continue processing.**

- *Note that the loop body does not display the last value assigned to product*

COMPOUND ASSIGNMENT OPERATORS

- C provides special assignment operators that enable a more concise notation for statements of this type. For the operations **+ , - , * , /** , and **%** , C defines the compound assignment operators **+= , -= , *= , /= ,** and **%=** .

TABLE 5.3 Compound Assignment Operators

Statement with Simple Assignment Operator	Equivalent Statement with Compound Assignment Operator
`count_emp = count_emp + 1;`	`count_emp += 1;`
`time = time - 1;`	`time -= 1;`
`total_time = total_time + times;`	`total time += time;`
`product = product * item;`	`product *= item;`
`n = n * (x + 1);`	`n *= x + 1;`

Exercise 4 p.247

s = s / 5;	1.	s /= 5;
q = q * n + 4;	2.	~~q *= n + 4;~~
z = z - x * y;	3.	z -= x * y;
t = t + (u % v);	4.	t += (u % v);

5.4 THE FOR STATEMENT

- **For statement** as another form for implementing loops.

- The loops we have seen so far are typical of most repetition structures in that they have three loop control components in addition to the loop body:

  - initialization of the *loop control variable,*
  - test of the *loop repetition condition,* and
  - change (update) of the *loop control variable.*

- **for statement** in C supplies a designated place for each of these three components

FIGURE 5.5 Using a 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);
```

- The effect of this for statement is exactly equivalent to the execution of the comparable while loop section of the program in Fig. 5.4.

- **For Statement Heading:** (1)

```
for (count_emp = 0;                    /* initialization Expression*/
count_emp < number_emp;          /* loop repetition condition */
count_emp += 1) {                /* update */
```

- The for statement can be used to count up or down by any interval.

➢ **Example:**

```
/* Display n asterisks. */
Printf(“Enter value for n”);
Scanf(“%d”, &n));
for (count_star = 0;
   count_star < n;
   count_star += 1)
      printf("*");
```

Caution: Although C permits the use of fractional values for counting loop control variables of type double , we strongly discourage this practice. (1)

INCREMENT AND DECREMENT OPERATORS

- The **increment operator** ++ takes a single variable as its operand.
- The side effect of applying the ++ operator is that the value of its operand is **incremented by one**.

- When the ++ is placed immediately in **front of** its operand (**prefix increment**), the value of the expression is the variable's value **after** incrementing:

for (counter = 0; counter < limit; **++counter**) (1)

- When the ++ comes immediately **after** the operand (**postfix increment**), the expression's value is the value of the variable **before** it is incremented.

for (counter = 0; counter < limit; **counter++**)

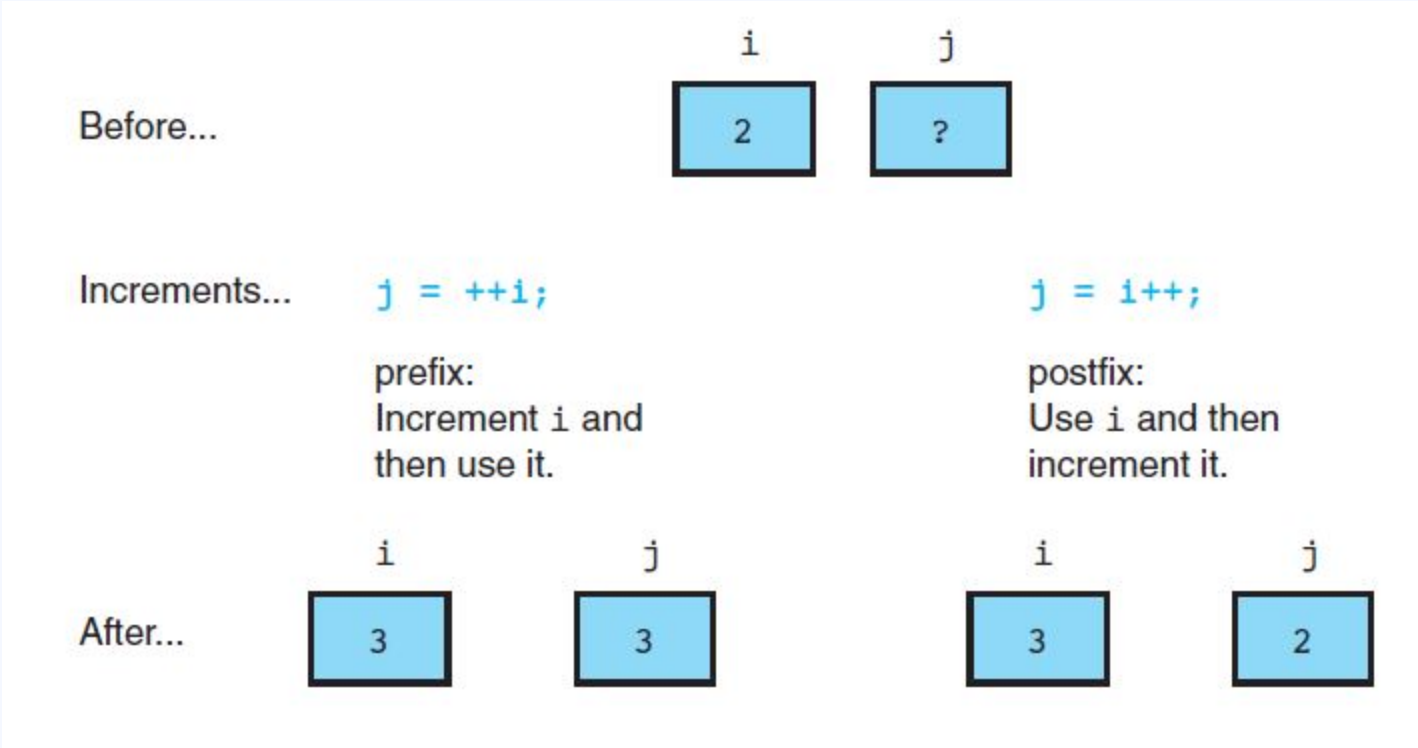


Figure 5.6

INCREMENT AND DECREMENT OPERATORS

- C also provides a **decrement operator** that can be used in either the prefix or postfix position.
- For example, if the initial value of n = 4

```
printf("%d", --n);    printf("%d", n);
```

3 3

```
printf("%d", n--);     printf("%d", n);
```

4 3

INCREMENT AND DECREMENT OPERATORS

- Avoid using the increment and decrement operators in complex expressions in which the variables to which they are applied appear more than once. (1)
- **For Example:**

```
x = 5;
i = 2;
y = i * x - ++i;
```

Y = 2 * 5 - 3 = 13 OR **Y = 3 * 5 - 3 = 18** ???

FIGURE 5.7 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. }
```

INCREMENTS AND DECREMENTS OTHER THAN 1

- We have seen for statement counting loops that count up **by one** and down by one.
- Now we will use a loop that counts down **by five** to display a Celsius-to-Fahrenheit conversion table.
- **EXAMPLE 5.4 p.251, figure 5.8**
- **Table 5.4 p.253 uses the small circled numbers to trace the execution of this counting for loop.**

5.5 CONDITIONAL LOOPS

- In many programming situations, you will not be able to determine the exact number of loop repetitions before loop execution begins.
- **For Example**: You want to continue prompting the user for a data value as long as the response is unreasonable:

Print an initial prompting message.

Get a positive number => Initialization Step

while the number is zero or negative => Loop repetition condition(1)

Print a warning and another prompting message.

Get a positive number => update step (2)

Note: Such a conditional loop still has three parts that control repetition(3)

This pattern is known as a **general conditional loop**.

we can write this validating input loop in C by using a while statement:

```
printf("Enter a positive number> ");
scanf("%d", &num);          /* initialization */
while (num <= 0) {          /* repetition condition */
    printf("Negative number entered; try again> ");
    scanf("%d", &num);      /* update */
}
```

5.6 GENERAL CONDITIONAL LOOP DESIGN

- **Sentinel-Controlled Loops:**
- Often we don't know how many data items the loop should process when it begins execution. Therefore, we must find some way to signal the program to stop reading and processing new data (1).

- One way to do this is to instruct the user to enter a unique data value, called a **sentinel value** , after the last data item.

- The loop repetition condition tests each data item and causes loop exit when the **sentinel value** is read.

- Choose the sentinel value carefully; it must be a value that could not normally occur as data.

SENTINEL-CONTROLLED LOOPS

- A loop that processes data until the sentinel value is entered has the form

  1. Get a line of data. => *initialization*
  2. while the **sentinel value** has not been encountered => *loop repetition condition*
     3. Process the data line.
     4. Get another line of data. => *update*

- For program readability, we usually name the sentinel by defining a **constant macro**

- **EXAMPLE 5.6 p.263**

USING A FOR STATEMENT TO IMPLEMENT A SENTINEL LOOP

- The for statement form of the while loop in **Fig. 5.10** follows:

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

ENDFILE-CONTROLLED LOOPS

- A data file is always terminated by an *endfile character* that can be detected by the *scanf* function.
- Therefore, you can write a batch program that processes a list of data of any length without requiring a special **sentinel value** at the end of the data.
- To write such a program, you must set up your input loop so it notices when *scanf* encounters the *endfile* character
- Besides storing new values in the variables passed to it as arguments, *scanf* also returns a result value just like the functions we studied.(1)

ENDFILE-CONTROLLED LOOPS

- input_status = scanf("%d%d%lf", &part_id, &num_avail, &cost);

- The function **scanf()** returns one of the following results:

1. Successful execution of the **scanf** in the values for the variables in its input list. (1)

2. if **scanf** runs into difficulty with invalid or insufficient data, the function returns as its value the number of data items scanned before encountering the error or running out of data (2)

3. Detecting the *endfile* character before getting input data for any of its arguments. In this case, **scanf** returns as its result the value of the standard constant ***EOF*** (**a negative integer**).

ENDFILE-CONTROLLED LOOPS

- It is possible to design a repetition statement very similar to the sentinel controlled loop that uses the status value returned by the scanning function to control repetition rather than using the values scanned.

- Pseudocode example for an *endfile-controlled loop*:

```
Get the first data value and save input status
while input status does not indicate that end of file has been reached
        Process data value
        Get next data value and save input status
```

Figure 5.11 page 267

INFINITE LOOPS ON FAULTY DATA

- The behavior of the **scanf** function when it encounters faulty data can quickly make infinite loops of the while statements in Fig. 5.10

- For example, let's assume the user responds to the prompt

  Enter next score (-99 to quit)> (1)

- In Fig. 5.11, Changing the loop repetition condition to

  input_status == 1

- would cause the loop to exit on either the end of file (input_status negative) or faulty data (input_status zero).(2)

INFINITE LOOPS ON FAULTY DATA

- We would also need to add an if statement after the loop to decide whether to simply print the results or to warn of bad input.

- The false task in the following if statement gets and displays the bad character when **input_status** is not EOF :

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

5.7 NESTED LOOPS

- Loops may be nested just like other control structures.
- Nested loops consist of an **outer loop** with one or more **inner loops**.
- Each time the outer loop is repeated, the inner loops are reentered, their loop control expressions are reevaluated, and all required iterations are performed.
- **Example 5.8, p.270** (1)

THE DO-WHILE STATEMENT

There are some situations, generally involving interactive input, when we know that a loop must execute at least one time.

We write the pseudocode for an input validation loop as follows:

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

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

```
/* Find first even number input */
do
     status = scanf("%d", &num);
while (status > 0    &&    (num % 2) != 0);
```

Note: If the loop body contains more than one statement, the group of statements must be **surrounded by braces**.

FLAG-CONTROLLED LOOPS FOR INPUT VALIDATION

- In many cases, the condition may be simplified by using a **flag**.
- A flag is a type **int** variable used to represent whether or not a certain event has occurred.
- A flag has one of two values**: 1 (true)** and **0 (false).**
- **See Example 5.10 p.274**

➢ The **do-while** is often the structure to choose when checking for valid input.

➢ The do-while used in Fig. 5.14 also prevents an infinite input loop in the event the user types an invalid character.

BREAK STATEMENTS

- Use **break** statements to exit out of a loop.
- End **break** statements with a semi-colon: **(break;)**

```
//stop when i = 3

for (int i = 0; i < 5; i++) {
    if (i == 3)
        break;
}
```

```
//get first odd number then stop
for (int i = start; i <= end; i++) {
    if (i % 2 != 0) { // Check if the number is odd

        printf("First odd number: %d\n", i);
        break;
    }
}
```

CONTINUE STATEMENTS

- Use **continue** statements to skip the current iteration of a loop and proceed with the next iteration without terminating the loop.
- End **continue** statements with a semi-colon: **(continue;)**

```
// Skip the iteration when i equals 3
for (int i = 0; i < 5; i++) {

    if (i == 3) {
        continue;
     }
     printf("i = %d\n", i);

}
```

```
//print only odd numbers
for (int i = start; i <= end; i++) {

        if (i % 2 == 0){ // Skip even numbers
 continue;
    }
   printf("Odd number: %d\n", i);
 }
```

5.10 HOW TO DEBUG AND TEST PROGRAMS

The first step in locating a hidden error is to examine the program output to determine which part of the program is generating incorrect results.

Then you can focus on the statements in that section of the program to determine which are at fault.

We describe two ways to do this:

- Using Debugger Programs
- Debugging without a Debugger

USING DEBUGGER PROGRAMS

A debugger program can help you debug a C program.

It lets you execute your program one statement at a time (single-step execution).

Through single-step execution:

You can **trace** your program's execution and observe the effect of each C statement on variables you select.

You can **validate** that loop control variables and other important variables (e.g., accumulators) are incremented as expected during each iteration of a loop.

You can also **check** that input variables contain the correct data after each ***scan*** operation.

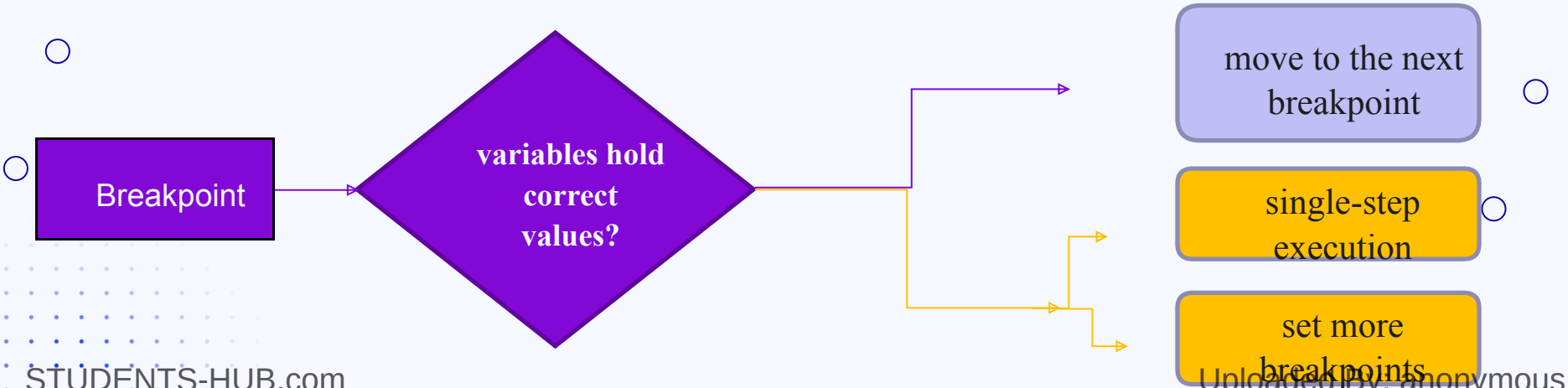
USING DEBUGGER PROGRAMS

Breakpoints:

If your program is very long, separate your program into segments by setting breakpoints at selected statements (1).

set a breakpoint at the end of each major algorithm step.

Then instruct the debugger to execute all statements from the last breakpoint up to the next breakpoint.



DEBUGGING WITHOUT A DEBUGGER [MANUAL DEBUGGING]

Insert extra diagnostic calls to **printf** that display intermediate results at critical points in your program.(1)

For example, you should display the values of variables affected by each major algorithm step before and after the step executes (2)

Once you have determined the likely source of an error, you should insert additional diagnostic calls to **printf** to trace the values of critical variables in the "buggy" segment.

DEBUGGING WITHOUT A DEBUGGER

- **Manual Debugging:**
- Turn diagnostics on by inserting: **#define DEBUG 1** (1)
- Turn diagnostics off by inserting: **#define DEBUG 0**

```
while (score != SENTINEL) {
        sum += score;
        if (DEBUG)
                printf("***** score is %d, sum is %d\n", score, sum);
        printf("Enter next score (%d to quit)> ", SENTINEL);
        scanf("%d", &score); /* Get next score. */
}
```

DEBUGGING WITHOUT A DEBUGGER

Include a **n** at the end of every **printf** format string. (1)

Be careful when you insert diagnostic calls to **printf** . Sometimes you must add braces if a single statement inside an if or a while statement becomes a compound statement when you add a diagnostic **printf** .

OFF-BY-ONE LOOP ERRORS

If a **sentinel-controlled** loop performs an extra repetition, it may erroneously process the sentinel value along with the regular data.

If a loop performs a counting operation, make sure that the initial and final values of the loop control variable are correct and that the loop repetition condition is right.

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

=>  for (count = 0; count < n; ++count)
```

OFF-BY-ONE LOOP ERRORS

Often you can determine whether a loop is correct by checking the **loop boundaries** —that is, the initial and final values of the loop control variable (1)

```
sum = 0; (2)
k = 1;
for (i = -n; i < n - k; ++i)
  sum += i * i;
```

When **i = -n** => **sum = n2** .

When **i = n - 2** , the value of **(n - 2)2** is added to the previous value of **sum**

TESTIN G

After all errors have been corrected and the program appears to execute as expected, the program should be tested thoroughly to make sure that it works.

For a simple program, make enough test runs to verify that the program works properly for representative samples of all possible data combinations.

5.12 COMMON PROGRAMMING ERRORS

Always use an **if** statement to implement a decision step and a **while** or **for** statement to implement a loop. (1) (2)

for (***initialization expression*****;**
loop repetition condition **;**
update expression **)**

End the initialization expression and the loop repetition condition with semicolons.

Do not put a semicolon before or after the closing parenthesis of the **for** statement header.(3)

5.12 COMMON PROGRAMMING ERRORS

Another common mistake in using **while** and **for** statements is to forget that the structure assumes that the loop body is a single statement. (1)

```
while (x > xbig)        (2)
   x -= 2;
   ++xbig;
/* intended end of while loop */
```

Executes As =>

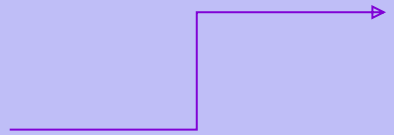
```
while (x > xbig)
   x -= 2; /* only this statement is repeated */
++xbig;
```

5.12 COMMON PROGRAMMING ERRORS

Error messages indicating a **missing closing brace** may appear at a different location than where the brace should be placed. (1)

When compound statements are nested, the compiler will associate the first closing brace encountered with the innermost structure. (2)

```
printf("Experiment successful? (Y/N)> ");
scanf("%c", &ans);
if (ans == 'Y') {
   printf("Enter one number per line (Enter %d to quit)\n> ", SENT);
   scanf("%d", &data);
   while (data != SENT) {
      sum += data;
      printf("> ");
      scanf("%d", &data);
   /* <— missing } */
} else {
   printf("Try it again tomorrow.\n");
   printf("Now follow correct shutdown procedure.\n");
}
```



5.12 COMMON PROGRAMMING ERRORS

```
scanf("%d%lf", &code, &amount);
while (balance != 0.0) {
    . . .
    scanf("%d%lf", &code, &amount);
}
```

=>

```
scanf("%d%lf", &code, &amount);
  while (balance > 0.0) {
   . . .
scanf("%d%lf", &code, &amount);
}
```

5.12 COMMON PROGRAMMING ERRORS

Be sure to verify that a loop's repetition condition will eventually become false (0) (1)

If you use a **sentinel-controlled loop**, remember provide a **prompt** that tells the program's user what value to enter as the sentinel (2)

One common cause of a nonterminating loop is the use of a loop repetition condition in which an **equality test** is mistyped as an **assignment operation**.

```
do {                    (3)
   . . .
   printf("One more time? (1 to continue/any other number to quit)> ");
   scanf("%d", &again);
} while (again = 1);        /* should be: again == 1 */
```

5.12 COMMON PROGRAMMING ERRORS

Use a **do-while** only when there is no possibility of zero loop iterations (1)

```
if ( condition 1 )        (2)
  do {
  . . .
} while ( condition 1 );
```

```
/*Replace the segment with a while or for stamenet */ (3)
while ( condition 1 ){
. . .
}
```

5.12 COMMON PROGRAMMING ERRORS

Remember the parentheses that are assumed to be around any expression that is the second operand of a compound assignment operator.

a *= b + c;

is equivalent to

a = a * (b + c);

there is no shorter way to write

a = a * b + c;

5.12 COMMON PROGRAMMING ERRORS

Do not use increment, decrement, or compound assignment operators as subexpressions in complex expressions.

Do not use a variable twice in an expression in which it is incremented/decremented. (1)

Be sure that the operand of an increment or decrement operator is a **variable** (2)

Refernces

Problem Solving and Program Design in C, 7th Ed., by Jeri R. Hanly and Elliot B. Koffman