C PROGRAM CONTROL – REPETITION

Deitel 8th Edition, Chapter 4

TOPICS

Repetition essentials

Parts of a loop

Types of Loops 1: Counter-Controlled

Implementing Counter-Controlled Loops with **while**

Compound Assignment Operators

Implementing Counter-Controlled Loops with **for**

Increment and Decrement Operators

TOPICS, P. 2

Common Errors & Other Observations when using **for**

Formatting Numeric Output

Implementing Counter-Controlled Loops with **do...while**

break and **continue** Statements

Types of Loops 2: Conditional Loops

Errors When Using **scanf** in Loops

REPETITION ESSENTIALS

TERMS

Repetition is also called **iteration**

Commonly called a **loop**

When is a Loop Needed?

When creating an algorithm, determine the **steps to solve one specific case**

Then ask:

- Do any of the steps for **this case** repeat?
- Does the whole algorithm need to be repeated for **more cases**?

If either answer is yes, then you need to implement a **loop**

ONCE YOU KNOW A LOOP IS REQUIRED…

You must ask:

- Is it known in advance how many times to repeat?
- If not, how do you know when to stop repeating?

There are different kinds of loops, based on the answers to the above questions.

SUMMARY OF THESE STEPS

STEP 1 – Use a loop if the algorithm – or any part of it – should **repeat**.

STEP 2 – Decide which kind of loop to use based on whether it is **known in advance how many times the repetition should occur**

- **STEP 2a** – if number of repetitions is known, use a **counter-controlled loop**
- **STEP 2b** – otherwise, use one of the **conditional loops**

All Loops Execute Based on the Value of a Condition

So all loops are **conditional loops**

But there are specific types of loops that are used in certain situations

Types of Repetition

Counter-controlled

- Repeat based on value of a counter variable

Sentinel-controlled

- Repeat based on value of a sentinel variable

Conditional

- Repeat based on the value of a condition, where a count or a sentinel is not involved

COUNTER-CONTROLLED REPETITION

Also called **definite repetition**

It's known before the loop runs how many times it will execute

Usually called a **counter-controlled loop**

Each execution of the loop (or, one pass through the loop) is called an **iteration**

SENTINEL-CONTROLLED REPETITION

Also called **indefinite** repetition

Not known in advance how many times loop will execute

Loop executes until a particular value is encountered

Usually called a **sentinel-controlled loop**

CONDITIONAL REPETITION

If a loop isn't counter- or sentinel-controlled, it's just called a conditional loop

Continues to execute until some situation exists

Remember, all loops are based on the value of a condition, so all loops are conditional.

- We just have some specific names for specific loops.

LOOP CONTROL VARIABLE

Part of the **condition**

Value determines whether the next iteration of the loop will occur

In counter-controlled repetition:

- Usually some kind of counter (integer or unsigned integer)

In sentinel-controlled repetition:

- Sentinel value indicates the end of the loop (usually the end of the data)

Parts of a Loop

All Loops are Based on the Value of a Condition

Remember the structure of a condition:

variable **conditional operator** **some value**

When working with loops, we give this variable a name:

Loop control variable

- Manages the repetitions of a loop

ALL LOOPS HAVE THREE PARTS

INITIALIZE the loop control variable to some starting value

TEST the loop control variable against some final value

- If the condition is true, execute loop body
- If the condition is false, stop executing loop

UPDATE the value of the loop control value

These may appear in different places in the actual loop structure, depending on the kind of loop.

Types of Loops 1: Counter-Controlled

COUNTER-CONTROLLED LOOPS

Loop whose **required number of iterations can be determined before loop execution begins**

So, it is known before the loop starts how many times the loop must repeat

Examples:

- Print monthly reports for the year
- Calculate grades for all students in class
- Ask the user for the number of calculations to perform
- Pump $5 worth of gas

COUNTER-CONTROLLED LOOPS GENERAL FORMAT

STEP 1: set loop control variable to 0

STEP 2: compare loop control variable to final value

If it is less than final value, proceed with loop body

If not, continue with code after loop

STEP 3: execute loop body

STEP 4: at end of loop body, add 1 to LCV

STEP 5: return to STEP 2

COUNTER-CONTROLLED LOOPS GENERAL FORMAT – STEPS IDENTIFIED

STEP 1: (**INITIALIZATION**) set loop control variable to 0

STEP 2: (**TEST**) compare loop control variable to final value

If it is less than final value, proceed with loop body

If not, continue with code after loop

STEP 3: execute loop body

STEP 4: (**UPDATE**) at end of loop body, add 1 to LCV

STEP 5: return to STEP 2

COUNTER-CONTROLLED LOOP EXAMPLE

STEP 1: (**INITIALIZATION**) set **dollars_spent** to 0

STEP 2: (**TEST**) compare **dollars_spent to 50**

If dollars_spent < 50, proceed with loop body

If not, continue with code after loop

STEP 3: add another dollar of gas

STEP 4: (**UPDATE**) at end of loop body, add 1 to dollars_spent

STEP 5: return to STEP 2

Implementing Counter-Controlled Loops with WHILE

THE WHILE STATEMENT

One way to implement counter-controlled repetition is to use the **while** statement

aka, a while loop

WHILE STATEMENT GENERAL FORMAT

Initialize loop control variable

while (compare lcv to final value**)**

{

loop body statement;

update lcv by adding 1

}

Requirements of Counter-Controlled Iteration

1. **Loop-control variable** (lcv) – counter
2. **INITIALIZATION** – Initial value for the lcv
3. **UPDATE** – a change in the lcv (usually an increment or decrement value)
4. **TEST** – a condition to test for final value of lcv

COUNTER-CONTROLLED ITERATION EXAMPLE

Fig04_01.c

Line 11 – increments the variable by 1

More on this shortly…

Floating-Point Values in Counter-Controlled Iteration

Floating-point values may be approximate

Controlling counting loops with floats or doubles may result in imprecise counter values and inaccurate termination tests.

Loop-control variables in counter-controlled repetition should always be integers

Compound Assignment Operators

COMMON TYPE OF ASSIGNMENT

What is the purpose of a statement like this?

count = count + 1;

This is such a common task, that there is shortcut syntax for writing it.

When to Use Compound Assignment Operators

When an assignment statement has this general format:

variable = same_variable operator value;

We can use a **compound assignment operator**:

variable operator= value;

compound assignment operator

COMPOUND ASSIGNMENT OPERATOR EXAMPLES

Statement with Simple Assignment Operator	Equivalent Statement with Compound Assignment Operator
count = count + 1;	count += 1;
time = time – 1;	time –= 1;
total_pay = total_pay + pay;	total_pay += pay;
product = product * item;	product *= item;

IMPLEMENTING COUNTER-CONTROLLED LOOPS WITH FOR

The for Statement

Remember the three loop control components:

1. **Initialization** of loop control variable
2. **Test** of loop repetition condition
3. **Update** of loop control variable

The for statement combines these into **one line**

FOR STATEMENT GENERAL FORMAT 1

```
for ( INITIALIZATION;  TEST;  UPDATE )
{
        loop body statement(s);
}
```

FOR STATEMENT GENERAL FORMAT 2

```
for ( INITIALIZATION;
      TEST;
      UPDATE )

{

      loop body statement(s)

}
```

FOR STATEMENT EXAMPLE 1

```
for ( x = 0;  x < max;  x = x + 1 )

    // loop body contains one statement
```

No curly braces needed if only one statement in body

one_line_body.c

FOR STATEMENT EXAMPLE 2

```
for ( x = 1; x < total;  x = x + 1)
{
        // statement 1
        // statement 2
        // etc.
}
```

Curly braces needed if more than one statement in body fig04_06.c

FOR STATEMENT EXAMPLE 3

```
for ( x = max;  x > 0;  x = x - 1)
    // stuff
```

Counting backwards

backwards.c

Increment and Decrement Operators

YET ANOTHER SYNTAX SHORTCUT

Remember this?

variable = same variable + a value

variable = same variable – a value

Which we shortened by using compound assignment operator:

variable += value

variable + – value

YET ANOTHER SYNTAX SHORTCUT – INCREMENT AND DECREMENT OPERATORS

If the value we're adding or subtracting is **1**:

variable = same variable + 1

variable = same variable – 1

We can use the **increment or decrement operator** instead:

variable++

variable – –

INCREMENT AND DECREMENT OPERATORS EXAMPLES 1

Assignment Statement	Equivalent Statement with Increment or Decrement Operator
count = count + 1;	count++;
lines = lines – 1;	lines– –;

INCREMENT AND DECREMENT OPERATORS EXAMPLES 2

Statement with Simple Assignment Operator	Equivalent Statement with Compound Assignment Operator	Equivalent Statement with Increment or Decrement Operator
count = count + 1;	count += 1;	count++;
time = time – 1;	time –= 1;	time – –;
salary = salary + OT;	salary += OT;	No equivalent
total = total * interest;	total *= interest;	No equivalent

PREVIOUS FOR STATEMENT EXAMPLE

```
for ( x = 1; x < total;  x = x + 1)
{
        // statement 1
        // statement 2
        // etc.
}
```

x = x + 1 can be replaced with?

To Make It More Interesting…

The increment or decrement operator can go before or after the variable name:

variable++;

variable – –;

++variable;

– – variable;

The operator works the same way, usually….

Side Effects

A change in the value of a variable as a result of carrying out an operation

Increment and decrement operators change the value of the operand

We have to be careful about using these operators in more complex expressions. It can be difficult to figure out what the value of the expression is.

EXAMPLE OF A SIDE EFFECT

What are the values of i, x and y after the following statements execute?

```
i = 5;

x = 2;

y = x * i++;
```

The value of an expression in which the ++ or – – operator is used depends on the operator's position within the expression.

PREFIX INCREMENT/DECREMENT OPERATORS

The ++ or – – is placed immediately before the operand:

++c

– – y

This creates an expression.

The value of this expression is the variable's value after incrementing.

PREFIX INCREMENT/DECREMENT OPERATORS – EXAMPLE 1

Line 1: **c = 10;**

Line 2: **++c;**

What is the value of the expression in line 2?

PREFIX INCREMENT/DECREMENT OPERATORS – EXAMPLE 1 RESULT

Line 1: **c = 10;**

Line 2: **++c;**

What is the value of the expression in line 2? **11**

PREFIX INCREMENT/DECREMENT OPERATORS – EXAMPLE 2

Line 1: **y = 10;**

Line 2: **– – y;**

What is the value of the expression in line 2?

PREFIX INCREMENT/DECREMENT OPERATORS – EXAMPLE 2 RESULT

Line 1: **y = 10;**

Line 2: **– – y;**

What is the value of the expression in line 2? **9**

POSTFIX INCREMENT/DECREMENT OPERATORS

The ++ or – – is placed immediately after the operand:

c++

y – –

This creates an expression.

The value of this expression is the variable's value after incrementing.

POSTFIX INCREMENT/DECREMENT OPERATORS – EXAMPLE 1

Line 1: **c = 10;**

Line 2: **c++;**

What is the value of the expression in line 2?

POSTFIX INCREMENT/DECREMENT OPERATORS – EXAMPLE 1 RESULT

Line 1: **c = 10;**

Line 2: **c++;**

What is the value of the expression in line 2? **11**

POSTFIX INCREMENT/DECREMENT OPERATORS – EXAMPLE 2

Line 1: **y = 10;**

Line 2: **y – – ;**

What is the value of the expression in line 2?

POSTFIX INCREMENT/DECREMENT OPERATORS – EXAMPLE 2 RESULT

Line 1: **y = 10;**

Line 2: **y – – ;**

What is the value of the expression in line 2? **9**

Rules for Evaluating Prefix and Postfix Operators

When pre- and postfix are used within **complex expressions**:

Prefix operator

- **Perform the increment or decrement BEFORE** evaluating the expression in which it appears
- "pre" = before

Postfix operator

- **Perform the increment or decrement AFTER** evaluating the expression in which it appears
- "post" = after

PREFIX AND POSTFIX OPERATORS COMPLEX EXPRESSION EXAMPLE 1, P. 1

What are the values of i, x and y after each of the following statements execute? (Assume each is initialized to 0 earlier.)

Line 1: **a = 5;**

Line 2: **x = 2;**

Line 3: **y = x * i++;**

Prefix and Postfix Operators Complex Expression Example 1, p. 2

What are the values of i, x and y after each of the following statements execute?

Line 1:	**a = 5;**	**a = 5, x = 0, y = 0**
Line 2:	**x = 2;**	**a = 5, x = 2, y = 0**
Line 3:	**y = x * i++;**	**a = 6, x = 2, y = ?**

a will end up as 6, because it is incremented by the ++

What will y be?

PREFIX AND POSTFIX OPERATORS COMPLEX EXPRESSION EXAMPLE 1, P. 3

The expression **i++** has the value 6, but because it is using a **POSTFIX** operator, the incrementing happens **AFTER** i is used in the larger expression. The original value of i is used in the multiplication, and i is incremented afterwards.

Line 1:	**a = 5;**	**a = 5, x = 0, y = 0**
Line 2:	**x = 2;**	**a = 5, x = 2, y = 0**
Line 3:	**y = x * i++;**	**a = 6, x = 2, y = 10**

pre_and_post_1.c

Prefix and Postfix Operators Complex Expression Example 2, p. 1

What are the values of a, x and y after each of the following statements execute? (Assume each is initialized to 0 earlier.)

Line 1: **a = 5;**

Line 2: **x = 2;**

Line 3: **y = x * ++a;**

Prefix and Postfix Operators Complex Expression Example 2, p. 2

What are the values of a, x and y after each of the following statements execute? (Assume each is initialized to 0 earlier.)

Line 1:	**a = 5;**	**a = 5, x = 0, y = 0**
Line 2:	**x = 2;**	**a = 5, x = 2, y = 0**
Line 3:	**y = x * ++a;**	**a = 6, x = 2, y = ?**

a will end up as 6, because it is incremented by the ++

What will y be?

PREFIX AND POSTFIX OPERATORS COMPLEX EXPRESSION EXAMPLE 2, P. 3

The expression **++a** has the value 6, but because it is using a **PREFIX** operator, the incrementing happens **BEFORE** a is used in the larger expression. The updated value of a is used in the multiplication.

Line 1:	**a = 5;**	**a = 5, x = 0, y = 0**
Line 2:	**x = 2;**	**a = 5, x = 2, y = 0**
Line 3:	**y = x * ++a;**	**a = 6, x = 2, y = 12**

pre_and_post_2.c

Operator Precedence

Partial list of operator precedence, with pre- and postfix operators:

Order	Type of Operator	Operators
1	Parentheses	**( )**
2	Postfix operators (L to R) Prefix operators (R to L)	**++ – –**
3	Binary operators (MDR)	*** / %**
4	Binary operators (AS)	**+ —**

COMMON ERRORS & OTHER OBSERVATIONS WHEN USING FOR

FOR **REPETITION STATEMENT – COMMON ERRORS**

Errors due to incorrect C standard use

Control variables defined in a **for header**

Off-by-one errors

ERRORS DUE TO INCORRECT C STANDARD

Example of another style of for loop:

```
for (int i = 0; i < 5; i++)

        printf( "The value of i is %d\n", i );
```

Note that the variable i is **declared and initialized** within the for header line

This isn't valid in all versions of C.

Errors Due to Incorrect C Standard – Error Message

The error message may look like this when the previous for loop is compiled under the wrong standard:

[Error] 'for' loop initial declarations are only allowed in C99 or C11 mode

[Note] use option -std=c99, -std=gnu99, -std=c11 or -std=gnu11 to compile your code

Errors Due to Incorrect C Standard – To Fix

To fix, change the standard used by the compiler in your IDE.

To fix in Dev-C++:

1. Tools menu
2. Compiler Options…
3. Settings tab
4. Code Generation tab
5. Language Standard – change to ISO C99

c_standard_error.c

Errors Related to Control Variables

Control variables defined in a **for header** exist only until the loop terminates

control_variables.c

We'll discuss this concept more next week.

OFF-BY-ONE ERRORS

Use the **final value** in the condition of a while or for statement and the **<=** relational operator to help avoid off-by-one errors.

off-by-one error.c

USING THE LCV WITHIN THE FOR LOOP BODY

Common to use the loop control variable for controlling repetition while never mentioning it in the body of the loop

Value of the LCV **can be changed** in the body of a **for** loop

- Can lead to subtle errors
- It's best not to change it

EXPRESSIONS IN THE FOR HEADER ARE OPTIONAL

If initialization expression (**initialization**) is omitted:

- LCV must be initialized before the for statement

If the condition expression (**test**) is omitted:

- Loop-continuation condition is assumed to be true, creating an infinite loop

The increment expression (**update**) may be omitted if:

- The increment is calculated by statements in the for statement's body, or if no increment is needed.

EXAMPLES USING THE FOR STATEMENT

Good examples to review on p. 120

Fig 4.5.c

Fig 4.6.c

- Includes math.h

FORMATTING NUMERIC OUTPUT

Formatting Numeric Output – Width, Precision, Justification

Indicate field width and precision values with a FCS in printf

Automatically right-justified if value to display is smaller than indicated field width

Indicate left justify with a minus

field_widths.c

DISPLAYING A TABLE OF VALUES

Note the printf in field_widths.c at line 19

printf("%6c%3d%8c%7.2f\n", ' ', celsius, ' ', fahrenheit);

The **space character enclosed in single quotes** is repeated, using the format placeholder %6c (repeats 6 times) and %8c (repeats 8 times)

IMPLEMENTING COUNTER-CONTROLLED LOOPS WITH DO...WHILE

DO...WHILE **ITERATION STATEMENT**

Loop-continuation statement executed at **end of loop**

This kind of loop **always executes at least once**

Often useful for displaying menus

Note the **%u** for displaying unsigned ints

Fig 4.9.c

BREAK **AND** CONTINUE **STATEMENTS**

BREAK **AND** CONTINUE **STATEMENTS ALTER TRANSFER OF CONTROL**

Statements used to alter the normal transfer of control:

Within a switch

Within a loop

BREAK **STATEMENT**

Causes an **immediate exit** from the **entire** while, do…while, for, or switch

Often used:

- To escape early from a loop
- To skip the rest of a switch (as we saw earlier)

4.11.c

- Loop fully executes 4 times
- Note the location of variable declaration

CONTINUE **STATEMENT**

Causes an **immediate exit** from the current **iteration** of while, do…while, or for

- Not used in switch

while and do loops

- Loop condition evaluated immediately after **continue**

for loops

- Update statement executes, then loop condition evaluated

Fig 4.12.c

Types of Loops 2: Conditional Loops

TYPES OF CONDITIONAL LOOPS

Conditional Loops

Sentinel-Controlled Loops

CONDITIONAL LOOPS

In many cases, we don't know how many times we have to execute a loop.

A **conditional loop** continues to execute **while** a condition is true

Most often implemented with **while** statement

CONDITIONAL LOOPS GENERAL FORMAT

Initialize loop control variable

while (compare lcv to some value**)**

{

loop body statement(s)

update lcv

}

CONDITIONAL LOOP EXAMPLE

Monitor experience points

Ask user for current amount of experience points, and amount used on quest

If there are enough points left, allow player to use some on a quest

Repeat

quest.c

SENTINEL-CONTROLLED LOOPS

Sentinel

- Signal to the program to stop the loop
- An end marker that follows the last item in a list of data

Sentinel-controlled loop

- Processes data until the sentinel value is entered

SENTINEL-CONTROLLED LOOP GENERAL FORMAT

Get some data from user

While the sentinel value has not been encountered

Process the data

Get more data from user

SENTINEL LOOP EXAMPLE

Calculate sum of exam scores entered by instructor. A score of -99 indicates the end of the scores.

scores.c

- Sum – accumulator variable
- Score – input variable
- Sentinel value is -99

What is the value of sum if the scores entered are 1, 2, 3, and then the sentinel -99?

Incorrect Syntax for a Sentinel Loop

Initialize sum to zero

While score is not the sentinel value

Get a score

Add score to sum

What is the value of sum if the scores entered are 1, 2, 3, and then the sentinel -99?

scores_incorrect_syntax.c

Errors When Using scanf in Loops

SCANF HAS A RETURN VALUE

scanf is a prebuilt **function** in C

Some **functions** (as we'll see next module) return some information to the user about what happened when the function executed

This information is called a **return value**

The return value from scanf is the **number of items successfully read** from the input stream

INFINITE LOOPS DUE TO FAULTY DATA & SCANF, P.1

Example: sentinel.c

If scanf is expecting %d, and the user types in 7o (number 7 followed by the letter o), the loop processes faulty data, resulting in an infinite loop.

The scanf reads the 7, but does not yet read the o (because it is only expecting numbers so far)

INFINITE LOOPS DUE TO FAULTY DATA & SCANF, P.2

The next loop iteration finds the letter o already waiting to be processed, so it tries to read it. But since o is not a valid integer, scanf returns zero (meaning nothing was read).

Since the loop condition doesn't use this return value, it's still waiting for an EOF

But the scanf keeps reading the letter o, resulting in an infinite loop

Use the return value to fix this…

sentinel_return_value.c