Introduction to Scientific and Engineering Computation (BIL 104E)

Lecture 4

Manipulating Data & Loops

Manipulating Data

The C language has a rich set of **operators** that let you manipulate data.

Arithmetic operators you have already learned are:

+ (**addition**) , **-** (**subtraction**) , ***** (**multiplication**) , **/** (**division**), and **%** (**remainder**)

You will learn more operators such as,

- **Arithmetic assignment operators**
- **Unary minus operator**
- **Increment and decrement operators**
- **Relational operators**
- **Cast operator**

Manipulating Data:
Arithmetic Assignment Operators

The Assignment Operator (=):
After the assignment, left-hand-operand will be equal to the value of right-hand-operand. The = operator always works from right to left.

The general statement form to use an assignment operator is

left-hand-operand = right-hand-operand;

The left-hand-operand must be a variable to receive data from the right-hand-operand .

For instance,

The statement **a = 5**; writes the value of the right-hand operand (**5**) into the memory location of the integer variable **a**.

The statement **b = a = 5**; assigns **5** to the integer variable **a** first, and then to the integer variable **b.** Now both **a** and **b** contain 5.

Manipulating Data:
Arithmetic Assignment Operators

Combining Arithmetic Operators with (=) :

By using the assignment operator (=) and the addition operator (+), you get the following statement:

z = x + y;

Now, write the result of the addition back to the integer variable, **x**

x = x + y;

The **(=)** operator always works from right to left, so the right side will be evaluated first then the previous value of **x** is replaced with the result of the addition from the right side.

(+=) is the new operator to do the addition and the assignment together.

x = x + y  **x += y;**

Manipulating Data: Arithmetic Assignment Operators

Arithmetic Assignment Operators:
consist of the combinations of the assignment operator (=) with the arithmetic operators, **+, -, *, /,** and **%**.

Operator	*Description*
+=	Addition assignment operator
-=	Subtraction assignment operator
*=	Multiplication assignment operator
/=	Division assignment operator
%=	Remainder assignment operator

the equivalence of statements:
`x += y;` is equivalent to `x = x + y;`
`x -= y;` is equivalent to `x = x - y;`
`x *= y;` is equivalent to `x = x * y;`
`x /= y;` is equivalent to `x = x / y;`
`x %= y;` is equivalent to `x = x % y;`

z = z * x + y; is not equivalent to the statement **z *= x + y;** because
z *= x + y multiplies **z** by the entire right-hand side of the statement, so the result would be the same as **z = z * (x + y);**

Manipulating Data: Arithmetic Assignment Operators

Using Arithmetic Assignment Operators

```
1:  /* 06L01.c: Using arithemtic assignment operators */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int x, y, z;
7:
8:     x = 1;   /* initialize x */
9:     y = 3;   /* initialize y */
10:    z = 10;  /* initialize z */
11:    printf("Given x = %d, y = %d, and z = %d,\n", x, y, z);
12:
13:    x = x + y;
14:    printf("x = x + y  assigns %d to x;\n", x);
15:
16:    x = 1;  /* reset x */
17:    x += y;
18:    printf("x += y  assigns %d to x;\n", x);
19:
20:    x = 1;  /* reset x */
21:    z = z * x + y;
22:    printf("z = z * x + y  assigns %d to z;\n", z);
23:
24:    z = 10;  /* reset z */
25:    z = z * (x + y);
26:    printf("z = z * (x + y) assigns %d to z;\n", z);
27:
28:    z = 10;  /* reset z */
29:    z *= x + y;
30:    printf("z *= x + y assigns %d to z.\n", z);
31:
32:    return 0;
33: }
```

```
Given x = 1, y = 3, and z = 10,
x = x + y  assigns 4 to x;
x += y  assigns 4 to x;
z = z * x + y  assigns 13 to z;
z = z * (x + y) assigns 40 to z;
z *= x + y assigns 40 to z.
```

Manipulating Data:
Unary Minus Operator

The minus operator (-) is used to change the sign of a number.

The (-) symbol is called the **unary minus operator**. Because the operator takes only one operand.

For instance:
given an integer of 7, you can get its negative value by changing the sign of the integer like -7

Don’t confuse the unary minus operator with the subtraction operator,
z = x - -y; is actually the same as this statement: **z = x - (-y);**

The first (-) symbol is used as the **subtraction operator**, while the second (-) symbol is the **unary minus operator.**

Manipulating Data:
Incrementing or Decrementing by One

The increment (++) and decrement (--) operators are used to add or subtract **1** from a variable.

For instance:

x = x + 1; statement is replaced with **++x;**
x = x - 1; statement is replaced with **--x**;

Pre-increment operator (++x):
Increment operator (++) appears before its operand and the operator first adds **1** to **x**, and then yields the new value of **x.**
Post-increment operator (x++):
Increment operator (++) appears after its operand and operand is used with old value in the calculation then operator adds 1 to **x**.

Manipulating Data:
Incrementing or Decrementing by One

Pre-decrement operator (--x):
Decrement operator (**--**) appears before its operand and the operator first substracts **1** from **x**, and then yields the new value of **x.**

Post-decrement operator (x--):
Decrement operator (**--**) appears after its operand and operand is used with old value in the calculation then operator substracts **1** from **x**.

For instance:

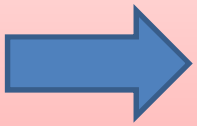
In the statement,

y = x++;

Before execution

x = 5

y = ?



After execution

x = 6

y = 5

y is assigned the original value of **x** first, then **x** is increased by **1**.

Manipulating Data: Incrementing or Decrementing by One

Using Pre- or Post-Increment and Decrement Operators

```
1:  /* 06L02.c: pre- or post-increment(decrement) operators */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int w, x, y, z, result;
7:
8:     w = x = y = z = 1;   /* initialize x and y */
9:     printf("Given w = %d, x = %d, y = %d, and z = %d,\n", w, x, y, z);
10:
11:    result = ++w;
12:    printf("++w evaluates to %d and w is now %d\n", result, w);
13:    result = x++;
14:    printf("x++ evaluates to %d and x is now %d\n", result, x);
15:    result = --y;
16:    printf("--y evaluates to %d and y is now %d\n", result, y);
17:    result = z--;
18:    printf("z-- evaluates to %d and z is now %d\n", result, z);
19:    return 0;
20: }
```

Manipulating Data: Incrementing or Decrementing by One

Computer Screen

```
Given w = 1, x = 1, y = 1, and z = 1,
++w evaluates to 2 and w is now 2
x++ evaluates to 1 and x is now 2
--y evaluates to 0 and y is now 0
z-- evaluates to 1 and z is now 0
```

Manipulating Data: Greater Than or Less Than?

There are six types of **relational operators** .

New operator used together with assignment operator should be left hand side of assignment operator.

Operator	*Description*
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Precedence property of relational operators:

All the **relational operators** have lower **precedence** than the **arithmetic operators**.

the **>, <, >=,** and **<= operators** have higher **precedence** than the **== and != operators**.

Manipulating Data: Greater Than or Less Than?

For instance:
the expression

x * y < z + 3 is interpreted as **(x * y) < (z + 3)**

Because the **arithmetic operators** have higher **presedence** than the **relational operators**.

A relational expression evaluates to **1** if the specified relationship is true. Otherwise, **0** is yielded.

For instance:
Given **x = 3** and **y = 5**,

The relational expression **x < y** that is **true** then gives a result of **1**.

Manipulating Data: Greater Than or Less Than?

Operator presedence:

Operator precedence refers to the order in which operators and operands are grouped together.

You can use parenthesis to group operands within an expression

For instance:

In the expression

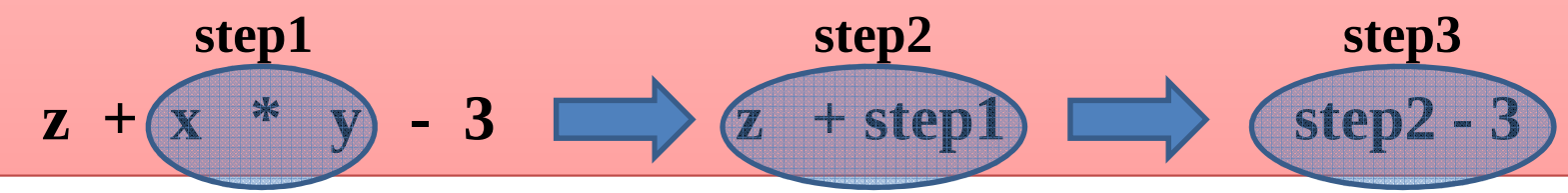
z + x * y - 3 → **(z + x) * (y - 3)**

The * operator has higher precedence than the + and - operators.

Step 1) **x * y** will be evaluated.

Step 2) The result of **step 1** becomes the right-hand operand of the + operator.

Step 3) The result of **step 2** is then given to the **-** operator as its lefthand operand.



Manipulating Data: Greater Than or Less Than?

Results Produced by Relational Expressions

```
1:  /* 06L03.c: Using relational operators */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int x, y;
7:     double z;
8:
9:     x = 7;
10:    y = 25;
11:    z = 24.46;
12:    printf("Given x = %d, y = %d, and z = %.2f,\n", x, y, z);
13:    printf("x >= y  produces: %d\n", x >= y);
14:    printf("x == y  produces: %d\n", x == y);
15:    printf("x < z   produces: %d\n", x < z);
16:    printf("y > z   produces: %d\n", y > z);
17:    printf("x != y - 18  produces: %d\n", x != y - 18);
18:    printf("x + y != z   produces: %d\n", x + y != z);
19:    return 0;
20: }
```

Manipulating Data: Greater Than or Less Than?

Computer Screen

```
Given x = 7, y = 25, and z = 24.46,
x >= y  produces: 0
x == y  produces: 0
x < z   produces: 1
y > z   produces: 1
x != y - 18  produces: 0
x + y != z   produces: 1
```

Manipulating Data: Using the Cast Operator

The data type of a variable, expression, or constant are converted to a different one by prefixing the **cast operator**.
This conversion does not change the operand itself.

The general form of the cast operator is

(data-type) x

data-type specifies the new data type.

For instance:

The expression

(float)5

converts the integer **5** to a floating-point number, **5.0**.

Manipulating Data: Using the Cast Operator

Playing with the Cast Operator

```
1:  /* 06L04.c: Using the cast operator */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int x, y;
7:
8:     x = 7;
9:     y = 5;
10:    printf("Given x = %d, y = %d\n", x, y);
11:    printf("x / y produces: %d\n",  x / y);
12:    printf("(float)x / y produces: %f\n",  (float)x / y);
13:    return 0;
14: }
```

```
Given x = 7, y = 5
x / y produces: 1
(float)x / y produces: 1.400000
```

Loops

Looping, also called **iteration**, is used in programming to perform the same set of statements over and over until certain specified conditions are met.

Three statements in C are designed for looping:

- ❖ The while statement
- ❖ The do-while statement
- ❖ The for statement

Loops: while

The purpose of the **while** keyword is to repeatedly execute a statement over and over while a given condition is true.

When the condition of the **while** loop is no longer logically true, the loop terminates and program execution resumes at the next statement following the loop.

The general form of the while statement is

```
while (expression)
    statement;
```

```
while (expression) {
    statement1;
    statement2;
}
```

Statement block

This process is repeated over and over until expression evaluates to zero, or logical false.

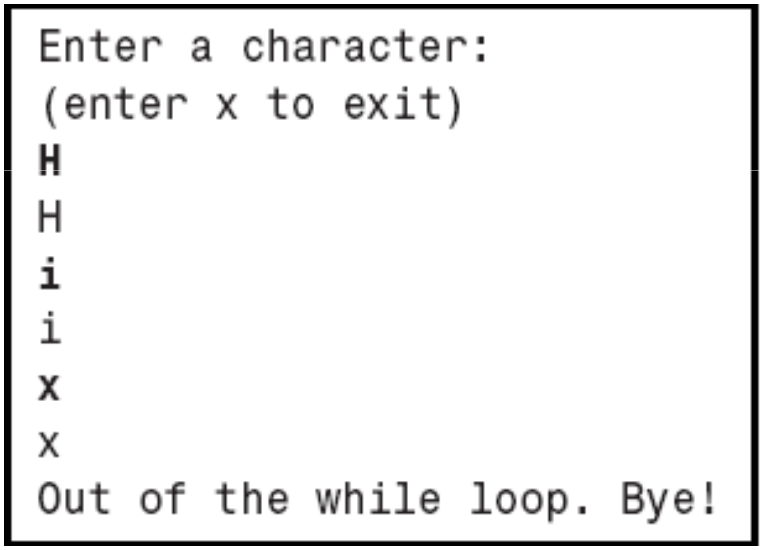
Loops: while

Using a while Loop

```
1:  /* 07L01.c: Using a while loop */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int c;
7:
8:     c = ' ';
9:     printf("Enter a character:\n(enter x to exit)\n");
10:    while (c != 'x') {
11:       c = getc(stdin);
12:       putchar(c);
13:    }
14:    printf("\nOut of the while loop. Bye!\n");
15:    return 0;
16: }
```

Loops: while

Computer Screen



Loops: do-while Loop

The statements inside the statement block are executed once, and then expression is evaluated in order to determine whether the looping is to continue.

If the expression evaluates to a nonzero value, the **do-while** loop continues; otherwise, the looping stops and execution proceeds to the next statement following the loop.

The general form for the do-while statement is

```
do {
        statement1;
        statement2;
        .
        .
} while (expression);
```

The **do-while** statement ends with a **semicolon**, which is an important distinction from the **if** and **while** statements.

Loops: do-while Loop

Using a do-while Loop

```
1:  /* 07L02.c: Using a do-while loop */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i;
7:
8:     i = 65;
9:     do {
10:       printf("The numeric value of %c is %d.\n", i, i);
11:       i++;
12:    } while (i<72);
13:    return 0;
14: }
```

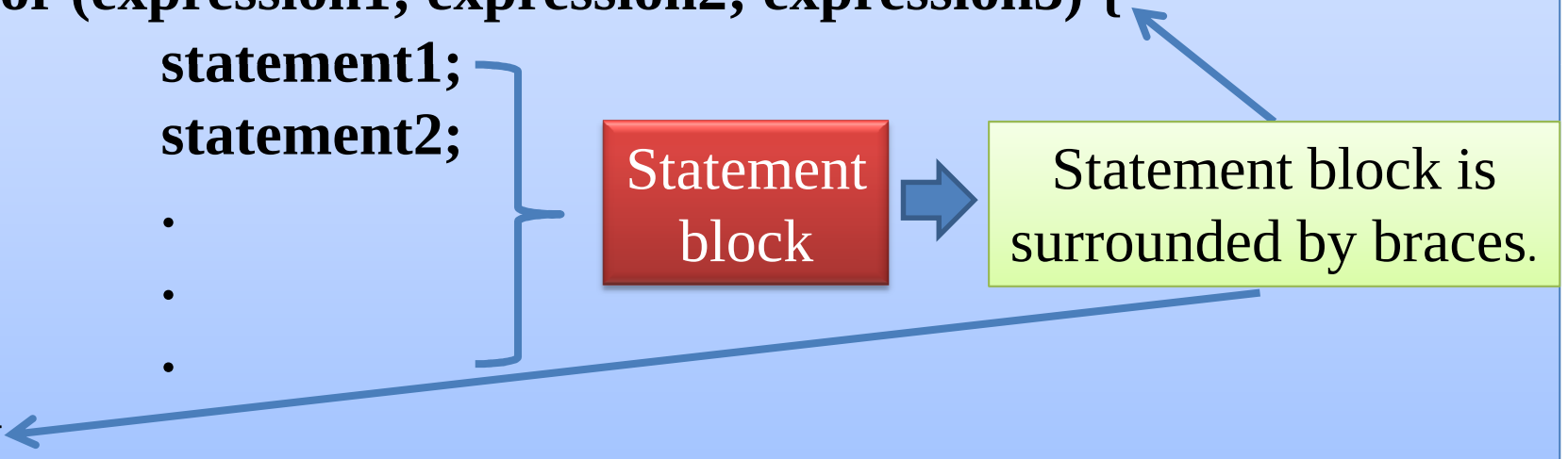
```
The numeric value of A is 65.
The numeric value of B is 66.
The numeric value of C is 67.
The numeric value of D is 68.
The numeric value of E is 69.
The numeric value of F is 70.
The numeric value of G is 71.
```

Loops: Looping Under the for Statement

The general form of the for statement is

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

for (expression1; expression2; expression3) {
        statement1;
        statement2;
        .
        .
        .
}
```



Loops: Looping Under the for Statement

expression1 is fistly evaluated, which is typically used to initialize one or more variables.

expression2 is evaluated immediately after **expression1.**
If **expression2** evaluates to a nonzero (logical true) value, the statements within the braces are executed.

expression3 is evaluated after each looping and before the statement goes back to test **expression2** again.

Loops: Looping Under the for Statement

Converting 0 through 15 to Hex Numbers

```
1:  /* 07L03.c: Converting 0 through 15 to hex numbers */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i;
7:
8:     printf("Hex(uppercase)   Hex(lowercase)   Decimal\n");
9:     for (i=0; i<16; i++){
10:       printf("%X                %x                %d\n", i, i, i);
11:    }
12:    return 0;
13: }
```

```
Hex(uppercase)   Hex(lowercase)   Decimal
0                0                0
1                1                1
2                2                2
3                3                3
4                4                4
5                5                5
6                6                6
7                7                7
8                8                8
9                9                9
A                a                10
B                b                11
C                c                12
D                d                13
E                e                14
F                f                15
```

Loops: The Null Statement

A **null statement** is a statement with no expression and contains nothing but a semicolon.

Syntax of null statement:

1) `for (i=0; i<8; i++);`

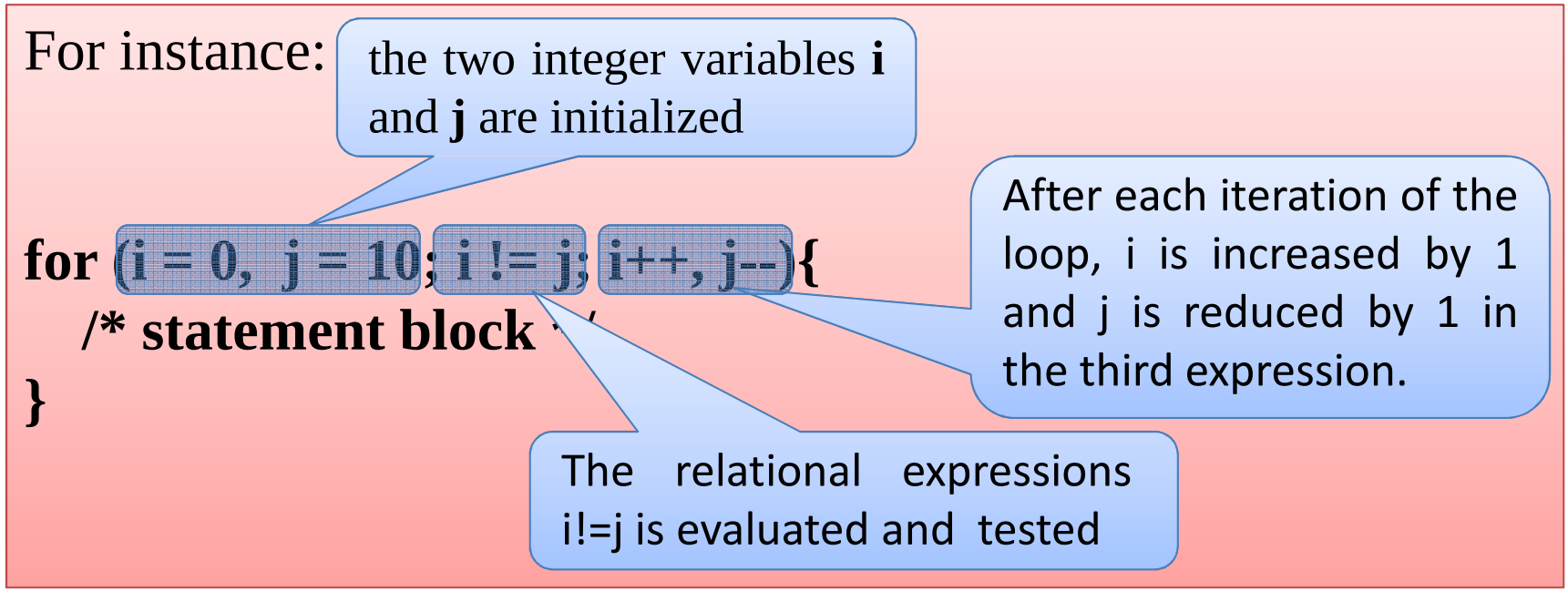
2)
```
for (i=0; i<8; i++)
        ;
```

After this loop, **i** is equal to **8.**

Loops: Using Complex Expressions in a for Statement

The different combinations of multiple expression are realized by the comma operator into the three parts of the for statement.

For instance:



Loops: Using Complex Expressions in a for Statement

Adding Multiple Expressions to the for Statement

```
1:  /* 07L04.c: Multiple expressions */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i, j;
7:
8:     for (i=0, j=8; i<8; i++, j--)
9:        printf("%d  +  %d  =  %d\n", i, j, i+j);
10:    return 0;
11: }
```

```
0  +  8  =  8
1  +  7  =  8
2  +  6  =  8
3  +  5  =  8
4  +  4  =  8
5  +  3  =  8
6  +  2  =  8
7  +  1  =  8
```

Loops: Using Complex Expressions in a for Statement

Another Example of Using Multiple Expressions in the for Statement

```
1:  /* 07L05.c: Another example of multiple expressions */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i, j;
7:
8:     for (i=0, j=1; i<8; i++, j++)
9:        printf("%d  -  %d  =  %d\n", j, i, j - i);
10:    return 0;
11: }
```

```
1  -  0  =  1
2  -  1  =  1
3  -  2  =  1
4  -  3  =  1
5  -  4  =  1
6  -  5  =  1
7  -  6  =  1
8  -  7  =  1
```

Loops: Using Nested Loops

It’s often necessary to create a loop even when you are already in a loop.

You can put a loop (an **inner loop**) inside another one (an **outer loop**) to make **nested loops**.

When the program reaches an inner loop, it will run just like any other statement inside the **outer loop**.

Loops: Using Nested Loops

Using Nested Loops

```
1:  /* 07L06.c: Demonstrating nested loops */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i, j;
7:
8:     for (i=1; i<=3; i++) {   /* outer loop */
9:        printf("The start of iteration %d of the outer loop.\n", i);
10:       for (j=1; j<=4; j++)  /* inner loop */
11:          printf("    Iteration %d of the inner loop.\n", j);
12:       printf("The end of iteration %d of the outer loop.\n", i);
13:    }
14:    return 0;
15: }
```

Loops: Using Nested Loops

Computer Screen

```
The start of iteration 1 of the outer loop.
    Iteration 1 of the inner loop.
    Iteration 2 of the inner loop.
    Iteration 3 of the inner loop.
    Iteration 4 of the inner loop.
The end of iteration 1 of the outer loop.
The start of iteration 2 of the outer loop.
    Iteration 1 of the inner loop.
    Iteration 2 of the inner loop.
    Iteration 3 of the inner loop.
    Iteration 4 of the inner loop.
The end of iteration 2 of the outer loop.
The start of iteration 3 of the outer loop.
    Iteration 1 of the inner loop.
    Iteration 2 of the inner loop.
    Iteration 3 of the inner loop.
    Iteration 4 of the inner loop.
The end of iteration 3 of the outer loop.
```