COMPUTER AIDED ENGINEERING LECTURE NOTES PART 2

MTE 201

**3. More Operators**

In this lesson you'll learn about more operators, including

1. Logical operators
2. The conditional operator

**Everything Is Logical**

Now, it's time for you to learn about a new set of operators: logical operators. There are three logical operators in the C language:

&& The logical AND operator

|| The logical OR operator

! The logical negation operator

The logical AND operator (&&) evaluates the truth or falseness of pairs of expressions. If both expressions are true, the logical AND operator returns 1. Otherwise, the operator returns 0.

However, the logical OR operator (||) returns 1 if at least one of the expressions is true. The || operator returns 0 if both expressions are false.

Only one operand (or expression) can be taken by the logical negation operator (!). If the operand is true, the ! operator returns 0; otherwise, the operator returns 1.

The following three sections contain examples that show you how to use the three logical operators. **The Logical AND Operator (&&)**

A general format of using the logical AND operator is:

exp1 && exp2

where exp1 and exp2 are two expressions evaluated by the AND operator. We can have a table that shows the return values of the AND operator under the following conditions when exp1 and

exp2 return 1 or 0, respectively. See Table 8.1, which can be called the truth table of the AND operator.

**Table 8.1. The values returned by the AND operator.**

exp1 exp2 Value Returned by &&

1 1 1

1 0 0

0 1 0

0 0 0

Listing 8.2 is an example of using the logical AND operator (&&).

**TYPE**

**Listing 8.2. Using the logical AND operator (&&).**

1: /\* 08L02.c: Using the logical AND operator \*/

2: #include <stdio.h>

3:

4: main()

5: {

6: int num;

7:

8: num = 0;

9: printf("The AND operator returns: %d\n",

10: (num%2 == 0) && (num%3 == 0));

11: num = 2;

12: printf("The AND operator returns: %d\n",

13: (num%2 == 0) && (num%3 == 0));

14: num = 3;

15: printf("The AND operator returns: %d\n",

16: (num%2 == 0) && (num%3 == 0));

17: num = 6;

18: printf("The AND operator returns: %d\n",

19: (num%2 == 0) && (num%3 == 0));

20:

21: return 0;

22: }

**OUTPUT**

After this program is compiled and linked, an executable file, 08L02.exe, is created. The following is the output printed on the screen after the executable is run from a DOS prompt on my machine:

C:\app> 08L02

The AND operator returns: 1

The AND operator returns: 0

The AND operator returns: 0

The AND operator returns: 1

C:\app>

**ANALYSIS**

In Listing 8.2, an integer variable, num, is declared in line 6 and initialized for the first time in line 8. Lines 9 and 10 print out the value returned by the logical AND operator in the following expression:

(num%2 == 0) && (num%3 == 0)

Here you see two relational expressions, num%2 == 0 and num%3 == 0. Previously, you learned that the arithmetic operator % can be used to obtain the remainder after its first operand is divided by the second operand. Therefore, num%2 yields the remainder of num divided by 2. The relational expression num%2 == 0 returns 1 (TRUE) if the remainder is equal to 0—that is, the value of num can be divided evenly by 2. Likewise, if the value of num can be divided by 3, the relational expression num%3 == 0 returns 1 as well. Then, according to the truth table of the && operator (see Table 8.1), we know that the combination of the logical AND operator (&&) and the two relational expressions yields 1 if the two relational expressions both return 1; otherwise, it yields 0. In our case, when num is initialized to 0 in line 8, both 0%2 and 0%3 yield remainders of 0 so that the two relational expressions return TRUE. Therefore, the logical AND operator returns 1. However, when num is assigned with the value of 2 or 3 as shown in lines 11 and 14, the logical AND operator in line 13 or line 16 returns 0. The reason is that 2 or 3 cannot be divided by both 2 and 3. Line 17 then assigns num the value of 6. Because 6 is a multiple of both 2 and 3, the logical AND operator in line 19 returns 1, which is printed out by the printf() function in lines 18 and 19.

From the program in Listing 8.2, you see several single statements spanning into multiple lines. The output from the program in Listing 8.2 shows the values returned by the AND operator when num is assigned with different values.

**The Logical OR Operator (||)**

As mentioned earlier, the logical OR operator returns 1 if at least one of the expressions is true. The || operator returns 0 if both expressions are false.

A general format of using the logical OR operator is:

exp1 || exp2

where exp1 and exp2 are two expressions evaluated by the OR operator.

**Table 8.2 shows the truth table of the OR operator.**

**exp1 exp2 Value Returned by ||**

1 1 1

1 0 1

0 1 1

0 0 0

The program in Listing 8.3 shows how to use the logical OR operator (||).

**TYPE**

**Listing 8.3. Using the logical OR operator (||).**

1: /\* 08L03.c: Using the logical OR operator \*/

2: #include <stdio.h>

3:

4: main()

5: {

6: int num;

7:

8: printf("Enter a single digit that can be divided\nby both 2 and 3:\n");

9: for (num = 1; (num%2 != 0) || (num%3 != 0); )

10: num = getchar() - 48;

11: printf("You got such a number: %d\n", num);

12: return 0;

13: }

**OUTPUT**

The following is the output printed on the screen after the executable, 08L03.exe, is run from a DOS prompt on my machine. The numbers in bold font are what I entered. (The Enter key is pressed after each number is entered.) In the range of 0\_9, 0 and 6 are the only two numbers that can be divided evenly by both 2 and 3:

C:\app> 08L03

Enter a single digit that can be divided

by both 2 and 3:

2

3

4

5

6

You got such a number: 6

C:\app>

**ANALYSIS**

In Listing 8.3, an integer variable, num, is declared in line 6. Line 8 of Listing 8.3 prints out a headline asking the user to enter a single digit. Note that there is a newline character (\n) in the middle of the headline message in the printf() function to break the message into two lines.

In line 9, the integer variable num is initialized in the first expression field of the for statement. The reason to initialize num with 1 is that 1 is such a number that cannot be divided by either 2 nor 3. Thus, the for loop is guaranteed to be executed at least once.

The key part of the program in Listing 8.3 is the logical expression in the for statement:

(num%2 != 0) || (num%3 != 0)

Here the relational expressions num%2 != 0 and num%3 != 0 are evaluated. According to the truth table of the || operator (see Table 8.2), we know that if one of the relational expression returns TRUE, i.e., the value of num cannot be divided completely by either 2 or 3. Then the logical expression returns 1, which allows the for loop to continue. The for loop stops only if the user enters a digit that can be divided by both 2 and 3. In other words, when both the relational expressions return FALSE, the logical OR operator yields 0, which causes the termination of the for loop.You can rewrite the program in Listing 8.3 with the if statement, too.

**The Logical Negation Operator (!)**

A general format of using the logical OR operator is:

!expression

where expression is an expression operated by the negation operator. The truth table of the negation operator is shown in Table 8.3.

**Table 8.3. The values returned by the ! operator.**

**Expression Value Returned by !**

1 0

0 1

**TYPE**

Now, let's take a look at the example, shown in Listing 8.4, that demonstrates how to use the logical negation operator (!).

**Listing 8.4. Using the logical negation operator (!).**

1: /\* 08L04.c: Using the logical negation operator \*/

2: #include <stdio.h>

3:

4: main()

5: {

6: int num;

7:

8: num = 7;

9: printf("Given num = 7\n");

10: printf("!(num < 7) returns: %d\n", !(num < 7));

11: printf("!(num > 7) returns: %d\n", !(num > 7));

12: printf("!(num == 7) returns: %d\n", !(num == 7));

13: return 0;

14: }

**OUTPUT**

The following result is obtained by running the executable file 08L04.exe:

C:\app> 08L04

Given num = 7

!(num < 7) returns: 1

!(num > 7) returns: 1

!(num == 7) returns: 0

C:\app>

**ANALYSIS**

In line 8, note that an integer variable, num, is initialized with 7, which is then displayed by the printf() function in line 9. In line 10, the relational expression num < 7 returns FALSE (that is, 0), because the value of num is not less than 7. However, by using the logical negation operator, !(num < 7) yields 1. (Refer to the truth table of the ! operator shown in Table 8.3.) Similarly, the logical expression !(num > 7) returns 1 in line 11. Because num has the value of 7, the relational expression num == 7 is true; however, the logical expression !(num == 7) in line 12 returns 0 due to the logical negation operator (!).

**What Does x?y:z Mean?**

In C, ?: is called the conditional operator, which is the only operator that takes three operands. The general form of the conditional operator is

x ? y : z

Here x, y, and z are three operands. Among them, x contains the test condition, and y and z represent the final value of the expression. If x returns nonzero (that is, TRUE), y is chosen; otherwise, z is the result. For instance, the expression x > 0 ? `T' : `F' returns `T' if the value of x is greater than 0. Otherwise, the expression returns `F'.

Listing 8.7 demonstrates the usage of the conditional operator in the C language.

**TYPE**

**Listing 8.7. Using the conditional operator.**

1: /\* 08L07.c: Using the ?: operator \*/

2: #include <stdio.h>

3:

4: main()

5: {

6: int x;

7:

8: x = sizeof(int);

9: printf("%s\n",

10: (x == 2) ? "The int data type has 2 bytes." : "int doesn't have 2

Âbytes.");

11: printf("The maximum value of int is: %d\n",

12: (x != 2) ? ~(1 << x \* 8 - 1) : ~(1 << 15) );

13: return 0;

14: }

**OUTPUT**

C:\app> 08L07

The int data type has 2 bytes.

The maximum value of int is: 32767

C:\app>

**ANALYSIS**

In Listing 8.7, the size of the int data type is measured first in line 8, by using the sizeof operator and the number of bytes assigned to the integer variable x. Lines 9 and 10 contain one statement, in which the conditional operator (?:) is used to test whether the number of bytes saved in x is equal to 2. (Here you see another example that a single statement can span multiple lines.) If the x == 2 expression returns nonzero (that is, TRUE), the string of The int data type has 2 bytes. is printed out by the printf() function in the statement. Otherwise, the second string, int doesn't have 2 bytes., is displayed on the screen. In addition, the statement in lines 11 and 12 tries to find out the maximum value of the int data type on the current machine. The x != 2 expression is evaluated first in the statement. If the expression returns nonzero (that is, the byte number of the int data type is not equal to 2), the ~( << x \* 8 - 1) expression is evaluated, and the result is chosen as the return value. Here the ~(1 << x \* 8 - 1) expression is a general form to calculate the maximum value of the int data type, which is equivalent to 2 \*\* (x \* 8 - 1) - 1.

On the other hand, if the test condition x != 2 in line 12 returns 0, which means the value of x is indeed equal to 2, the result of the ~(1 << 15) expression is chosen. Here you may have already figured out that ~(1 << 15) is equivalent to 215\_1, which is the maximum value that the 16-bit int data type can have. The result displayed on the screen shows that the int data type on my machine is 2 bytes (or 16 bits) long, and the maximum value of the int data type is 32767.

**4. LOOPING**

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:

1. The for statement
2. The while statement
3. The do-while statement

**Looping Under the for Statement**

The general form of the for statement is

for (expression1; expression2; expression3) {

statement1;

statement2;

.

.

.

}

You see from this example that the for statement uses three expressions (expression1, expression2, and expression3) that are separated by semicolons.

Several statements, such as statement1 and statement2, are placed within the braces ({ and }). All the statements and the braces form a statement block that is treated as a single statement.

In the preceding for statement format, the beginning brace ({) is put on the same line of the for keyword. You can place the beginning brace on a separate line beneath the for keyword.

The for statement first evaluates expression1, which usually initializes one or more variables. In other words, expression1 is only evaluated once when the for statement is first encountered.

The second expression, expression2, is the conditional part that is evaluated right after the evaluation of expression1 and then is evaluated after each successful looping by the for statement. If expression2 returns a nonzero value, the statements within the braces are executed. Usually, the nonzero value is 1. If expression2 returns 0, the looping is stopped and the execution of the for statement is finished.

The third expression in the for statement, expression3, is not evaluated when the for statement is first encountered. However, expression3 is evaluated after each looping and before the statement goes back to test expression2 again.

**Listing 7.1. Converting 0 through 15 to hex numbers.**

1: /\* 07L01.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: }

**OUTPUT**

C:\app> 07L01

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

C:\app>

**ANALYSIS**

Now, let's have a look at the code in Listing 7.1. As you know, line 2 includes the header file stdio.h for the printf() function used later in the program.

Inside the body of the main() function, the statement in line 6 declares an integer variable, i. Line 8 displays the headline of the output on the screen.

Lines 9\_11 contain the for statement. Note that the first expression in the for statement is i=0, which is an assignment expression that initializes the integer variable i to 0.

The second expression in the for statement is i<16, which is a relational expression. This expression returns 1 as long as the relation indicated by the less-than operator (<) holds. As mentioned earlier, the second expression is evaluated by the for statement each time after a successful looping. If the value of i is less than 16, which means the relational expression remains true, the for statement will start another loop. Otherwise, it will stop looping and exit.

The third expression in the for statement is i++ in this case. This expression is evaluated and the integer variable i is increased by 1 each time after the statement inside the body of the for statement is executed. Here it doesn't make a big difference whether the post-increment operator (i++) or the pre-increment operator (++i) is used in the third expression.

In other words, when the for loop is first encountered, i is set to 0, the expression i<16 is evaluated and found to be true, and therefore the statements within the body of the for loop are executed. Following execution of the for loop, the third expression i++ is executed incrementing i to 1, and i<16 is again evaluated and found to be true, thus the body of the loop is executed again. The looping lasts until the conditional expression i<16 is no longer true.

There is only one statement inside the for statement body, as you can see in line 10. The statement contains the printf() function, which is used to display the hex numbers (both uppercase and lowercase) converted from the decimal values by using the format specifiers, %X and %x.

Here the decimal value is provided by the integer variable i. As explained, i contains the initial value of 0 right before and during the first looping. After each looping, i is increased by 1 because of the third expression, i++, in the for statement. The last value provided by i is 15. When i reaches 16, the relation indicated by the second expression, i<16, is no longer true. Therefore, the looping is stopped and the execution of the for statement is completed.

Then, the statement in line 12 returns 0 to indicate a normal termination of the program, and finally, the main() function ends and returns the control back to the operating system.

As you see, with the for statement, you can write a very concise program. Actually, you can make the program in Listing 7.1 even shorter. In the for statement, you can discard the braces ({ and }) if there is only one statement inside the statement block.

**The Null Statement**

As you may notice, the for statement does not end with a semicolon. The for statement has within it either a statement block that ends with the closing brace (}) or a single statement that ends with a semicolon. The following for statement contains a single statement:

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

sum += i;

Now consider a statement such as this:

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

Here the for statement is followed by a semicolon immediately. In the C language, there is a special statement called the null statement. A null statement contains nothing but a semicolon. In other words, a null statement is a statement with no expression.

Therefore, when you review the statement for (i=0; i<8; i++);, you can see that it is actually a for statement with a null statement. In other words, you can rewrite it as

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

;

Because the null statement has no expression, the for statement actually does nothing but loop.

**Adding More Expressions into for**

The C language allows you to put more expressions into the three expression fields in the for statement. Expressions in a single expression field are separated by commas.

For instance, the following form is valid in C:

for (i=0, j=10; i<10, j>0; i++, j--){

/\* statement block \*/

}

Here, in the first expression field, the two integer variables, i and j, are initialized, respectively, with 0 and 10 when the for statement is first encountered. Then, in the second field, the two relational expressions, i<10 and j>0, are evaluated and tested. If one of the relational expressions returns 0, the looping is stopped. After each iteration and the statements in the statement block are executed successfully, i is increased by 1, j is reduced by 1 in the third expression field, and the expressions i<10 and j>0 are evaluated to determine whether to do one more looping.

**The while Loop**

The while statement is also used for looping. Unlike the situation with the for statement, there is only one expression field in the while statement.

The general form of the while statement is

while (expression) {

statement1;

statement2;

.

.

.

}

Here expression is the field of the expression in the while statement. The expression is evaluated first. If the expression returns a nonzero value (normally 1), the looping continues; that is, the statements inside the statement block are executed. After the execution, the expression is evaluated again. The statements are then executed one more time if the expression still returns nonzero value. The process is repeated over and over until the expression returns 0.

You see that a statement block, surround by the braces { and }, follows the while keyword and the expression field. Of course, if there is only one statement in the statement block, the braces can be discarded.

**TYPE**

**Listing 7.5. Using a while loop.**

1: /\* 07L05.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: }

**OUTPUT**

The executable 07L05.exe can do a similar job as the executable 07L04.exe. The following is a copy from the screen:

C:\app> 07L05

Enter a character:

(enter x to exit)

H

H

i

i

x

x

Out of the while loop. Bye!

C:\app>

**ANALYSIS**

You see that the output from the execution of the program in Listing 7.5 is similar to the one from Listing 7.4, except the while statement is used in lines 10\_13 of Listing 7.5.

The char variable c is initialized with a space character in line 8. Unlike the for statement in Listing 7.4, the while statement does not set c before the looping.

In line 10, the relational expression c != `x' is tested. If the expression returns 1, which means the relation still holds, the statements in lines 11 and 12 are executed. The looping continues as long as the expression returns 1. If, however, the user enters the character x, which makes the relational expression return 0, the looping stops.

**The do-while Loop**

You may note that in the for and while statements, the expressions are set at the top of the loop. However, in this section, you're going to see another statement used for looping, do-while, which puts the expressions at the bottom of the loop. In this way, the statements controlled by the do-while statement are executed at least once before the expression is tested. Note that statements in a for or while loop are not executed at all if the condition expression does not hold in the for or while statement.

The general form for the do-while statement is

do {

statement1;

statement2;

.

.

.

} while (expression);

Here expression is the field for the expression that is evaluated in order to determine whether the statements inside the statement block are to be executed one more time. If the expression returns a nonzero value, the do-while loop continues; otherwise, the looping stops.

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

The program in Listing 7.6 displays the characters A through G by using a do-while loop to repeat the printing and adding.

**TYPE**

**Listing 7.6. Using a do-while loop.**

1: /\* 07L06.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: }

**OUTPUT**

After running the executable 07L06.exe of Listing 7.6, I have the characters A through G, along with their numeric values, shown on the screen as follows:

C:\app> 07L06

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.

C:\app>

**ANALYSIS**

The statement in line 8 of Listing 7.6 initializes the integer variable i with 65. The integer variable is declared in line 6.

Lines 9\_12 contain the do-while loop. The expression i<72 is at the bottom of the loop in line 12. When the loop first starts, the two statements in lines 10 and 11 are executed before the expression is evaluated. Because the integer variable i contains the initial value of 65, the printf() function in line 10 displays the numeric value as well as the corresponding character A on the screen. After the integer variable i is increased by 1 in line 11, the program control reaches the bottom of the do-while loop. Then the expression i<72 is evaluated. If the relationship in the expression still holds, the program control jumps up to the top of the do-while loop, and then the process is repeated. When the expression returns 0 after i is increased to 72, the do-while loop is terminated immediately.

QUIZ

1. How does a for loop work?
2. What is the difference between the while and do-while statements?
3. Can the while statement end with a semicolon?

Do the following two for loops have the same number of iterations?

for (j=0; j<8; j++);

for (k=1; k<=8; k++);

1. Is the following for loop

for (j=65; j<72; j++) printf("%c", j);

int k = 65;

while (k<72)

printf("%c", k);

k++;

}

1. Can the following while loop print out anything?

int k = 100;

while (k<100){

printf("%c", k);

k++;

}

1. Can the following do-while loop print out anything?

int k = 100;

do {

printf("%c", k);

k++;

} while (k<100);

1. What is the difference between the following two pieces of code?

for (i=0, j=1; i<8; i++, j++)

printf("%d + %d = %d\n", i, j, i+j);

for (i=0, j=1; i<8; i++, j++);

printf("%d + %d = %d\n", i, j, i+j);

1. Rewrite the program in Listing 7.4. This time, you want the for statement to keep looping until the user enters the character K.
2. Rewrite the program in Listing 7.6 by replacing the do-while loop with a for loop.
3. Rewrite the program in Listing 7.7. This time, use a while loop as the outer loop and a do-while loop as the inner loop.
4. **CONTROLS AND DECISION MAKING**

In this lesson you'll learn about the statements that belong to another group of control flow statements—conditional branching (or jumping), such as

The if statement

The if-else statement

The switch statement

The break statement

The continue statement

The goto statement

The general form of the **if** statement is

if (expression) {

statement1;

statement2;

.

.

.

}

Here expression is the conditional criterion. If expression is logical TRUE (that is, nonzero), the statements inside the braces ({ and }), such as statement1 and statement2, are executed. If expression is logical FALSE (zero), then the statements are skipped.

Note that the braces ({ and }) form a block of statements that is under the control of the if statement. If there is only one statement inside the block, the braces can be ignored. The parentheses (( and )), however, must always be used to enclose the conditional expression.

For instance, the following expression

if (x > 0)

printf("The square root of x is: %f\n", sqrt(x));

tells the computer that if the value of x is greater than zero (that is, positive), it should calculate the square root of x by calling the sqrt() function, and then print out the result. Here the conditional criterion is the relational expression x > 0, which returns 1 for true and 0 for false.

**TYPE**

**Listing 10.1. Using the if statement in decision making.**

1: /\* 10L01.c Using the if statement \*/

2: #include <stdio.h>

3:

4: main()

5: {

6: int i;

7:

8: printf("Integers that can be divided by both 2 and 3\n");

9: printf("(within the range of 0 to 100):\n");

10: for (i=0; i<=100; i++){

11: if ((i%2 == 0) && (i%3 == 0))

12: printf(" %d\n", i);

13: }

14: return 0;

15: }

**OUTPUT**

After 10L01.exe, the executable of the program in Listing 10.1, is created and run from a DOS prompt, the following output is printed on the screen:

C:\app> 10L01

Integers that can be divided by both 2 and 3

(within the range of 0 to 100):

0

6

12

18

24

30

36

42

48

54

60

66

72

78

84

90

96

C:\app>

**ANALYSIS**

As you see in Listing 10.1, line 6 declares an integer variable, i. Lines 8 and 9 print out two headlines. Starting in line 10, the for statement keeps looping 101 times.

Within the for loop, the if statement in lines 11 and 12 evaluates the logical expression (i%2 == 0) && (i%3 == 0). If the expression returns 1 (that is, the value of i can be divided by both 2 and 3 completely), the value of i is displayed on the screen by calling the printf() function in line 12. Otherwise, the statement in line 12 is skipped.

Note that the braces ({ and }) are discarded in the if statement because there is only one statement under the control of the statement.

The result shown on the screen gives all integers within the range of 0 to 100 that can be divided by both 2 and 3.

**The if-else Statement**

In the if statement, when the conditional expression is logical TRUE, the computer will jump to the statements controlled by the if statement and execute them right away. If the expression is false, the computer will ignore those statements controlled by the if statement.

From time to time, you will want the computer to execute some other specified statements when the conditional expression of the if statement is logical FALSE. By doing so, you can use another conditional branching statement in C—the if-else statement.

As an expansion of the if statement, the if-else statement has the following form:

if (expression) {

statement1;

statement2;

.

.

.

}

else {

statement\_A;

statement\_B;

.

.

.

}

Here if expression is logical TRUE, the statements controlled by if, including statement1 and statement2, are executed. The statements, such as statement\_A and statement\_B, inside the statement block and following the else keyword are executed if expression is not logical TRUE.

The program in Listing 10.2 shows how to use the if-else statement.

**TYPE**

**Listing 10.2. Using the if-else statement.**

1: /\* 10L02.c Using the if-else statement \*/

2: #include <stdio.h>

3:

4: main()

5: {

6: int i;

7:

8: printf("Even Number Odd Number\n");

9: for (i=0; i<10; i++)

10: if (i%2 == 0)

11: printf("%d", i);

12: else

13: printf("%14d\n", i);

14:

15: return 0;

16: }

**OUTPUT**

The following result is obtained by running the executable file 10L02.exe:

C:\app>10L02

Even Number Odd Number

0 1

2 3

4 5

6 7

8 9

C:\app>

**ANALYSIS**

Line 6 of Listing 10.2 declares an integer variable, i. The printf() function in line 8 displays a headline on the screen. The integer variable i is initialized in the first expression field of the for statement in line 9. Controlled by the for statement, the if-else statement in lines 10\_13 is executed 10 times. According to the if-else statement, the printf() function in line 11 prints out even numbers if the relational expression i%2 == 0 in line 10 returns 1 (that is, TRUE).

If the relational expression returns 0 (that is, FALSE), the printf() function controlled by the else keyword in line 12 outputs odd numbers to the standard output. Because the if-else statement is treated as a single statement, the braces { and } are not needed to form a block of statement in the for statement. Likewise, there are no braces used in the if-else statement because the if and else keywords each control a single statement, respectively, in lines 11 and 13. Note that the minimum width of 14 is specified in the printf() function in line 13, so the output of the odd numbers is listed to the right side of the even numbers, as you can see in the output section. Because the program in Listing 10.2 checks numbers in a range of 0 to 9, the output shows that 0, 2, 4, 6, and 8 are even numbers, and 1, 3, 5, 7, and 9 are odd ones.

**Nested if Statements**

As you saw in the previous sections, one if statement enables a program to make one decision. In many cases, a program has to make a series of decisions. To enable it to do so, you can use nested if statements. Listing 10.3 demonstrates the usage of nested if statements.

**TYPE**

**Listing 10.3. Using nested if statements.**

1: /\* 10L03.c Using nested if statements \*/

2: #include <stdio.h>

3:

4: main()

5: {

6: int i;

7:

8: for (i=-5; i<=5; i++){

9: if (i > 0)

10: if (i%2 == 0)

11: printf("%d is an even number.\n", i);

12: else

13: printf("%d is an odd number.\n", i);

14: else if (i == 0)

15: printf("The number is zero.\n");

16: else

17: printf("Negative number: %d\n", i);

18: }

19: return 0;

20: }

**OUTPUT**

After running the executable file 10L03.exe, I obtain the following output:

C:\app>10L03

Negative number: -5

Negative number: -4

Negative number: -3

Negative number: -2

Negative number: -1

The number is zero.

1 is an odd number.

2 is an even number.

3 is an odd number.

4 is an even number.

5 is an odd number.

C:\app>

**ANALYSIS**

Listing 10.3 contains a for loop, starting in line 8 and ending in line 18. According to the expressions of the for statement in line 8, any tasks controlled by the for statement are executed up to 11 times. First, a decision has to be made based on the return value of the relational expression i > 0 in the if statement of line 9. The i > 0 expression is used to test whether the value of i is positive or negative (including zero.) If the return value is 1, the computer jumps to the second (that is, nested) if statement in line 10. Note that line 10 contains another relational expression, i%2 == 0, which tests whether the integer variable i is even or odd. Therefore, the second decision of displaying even numbers or odd numbers has to be made according to the

return value of the second relational expression, i%2 == 0. The printf() function in line 11 prints out an even number if the return value is 1. Otherwise, the statement in line 13 is executed, and an odd number is shown on the screen. The computer branches to line 14 if the i > 0 expression returns 0; that is, if the value of i is not greater than 0. In line 14, another if statement is nested within an else phrase, and the relational expression i == 0 is evaluated. If i == 0 is true, which means i contains the value of zero, the string of The number is zero. is displayed on the screen.

Otherwise, the value of i is negative, according to the value returned by the i > 0 expression. The statement in line 17 then outputs the negative number to the standard output.

As you can see in the example, the value of i is within the range of 5 to -5. Thus, -5, -4, -3, -2, and -1 are printed out as negative numbers. In addition, the odd numbers 1, 3, and 5, as well as the even numbers 2 and 4, are also printed out.

**The switch Statement**

In the last section, you saw that nested if statements are used when there is more than one decision to be made. The nested if statements will become very complex if there are many decisions that need to be made, however. Sometimes, the programmer will have problems following the complex if statements. Fortunately there is another statement in C, the switch statement, that you can use to make unlimited decisions or choices based on the value of a conditional expression and specified cases. The general form of the switch statement is

switch (expression) {

case expression1:

statement1;

case expression2:

statement2;

.

.

.

default:

statement-default;

}

Here the conditional expression, expression, is evaluated first. If the return value of expression is equal to the constant expression expression1, the statement statement1 is executed. If the value of expression is the same as the value of expression2, statement2 is executed. If, however, the value of expression is not equal to any values of the constant expressions labeled by the case keyword, the statement (statement-default) following the default keyword is executed. You have to use the case keyword to label each case. The default keyword is recommended to be used for the default case. Note that no constant expressions are identical in the switch statement.

The program in Listing 10.4 gives you an example of using the switch statement. The program also demonstrates an important feature of the switch statement.

**TYPE**

**Listing 10.4. Using the switch statement.**

1: /\* 10L04.c Using the switch statement \*/

2: #include <stdio.h>

3:

4: main()

5: {

6: int day;

7:

8: printf("Please enter a single digit for a day\n");

9: printf("(within the range of 1 to 3):\n");

10: day = getchar();

11: switch (day){

12: case `1':

13: printf("Day 1\n");

14: case `2':

15: printf("Day 2\n");

16: case `3':

17: printf("Day 3\n");

18: default:

19: ;

20: }

21: return 0;

22: }

**OUTPUT**

C:\app>10L04

Please enter a single digit for a day

(within the range of 1 to 3):

3

Day 3

C:\app>

**ANALYSIS**

As you can see in line 6, an int variable, day, is declared; it is assigned the input entered by the user in line 10.

In line 11, the value of the integer variable day is evaluated in the switch statement. If the value is equal to one of the values of the constant expressions, the computer starts to execute statements from there. The constant expressions are labeled by prefixing case in front of them.

For instance, I entered 3 and then pressed the Enter key. The numeric value of 3 is assigned to day in line 10. Then, after finding a case in which the value of the constant expression matches the value contained by day, the computer jumps to line 17 to execute the printf() function and display Day 3 on the screen. Note that under the default label in Listing 10.4, there is an empty (that is, null) statement ending with semicolon (;) in line 19. The computer does nothing with the empty statement. However, if I enter 1 from my keyboard and then press the Enter key, I get the following output:

C:\app>10L04

Please enter a single digit for a day

(within the range of 1 to 3):

1

Day 1

Day 2

Day 3

C:\app>

**OUTPUT**

From the output, you can see that the statement controlled by the selected case, case 1, and the statements controlled by the rest of the cases are executed, because Day 1, Day 2, and Day 3 are displayed on the screen. Likewise, if I enter 2 from my keyboard, I have Day2 and Day3 shown on the screen. This is an important feature of the switch statement: The computer continues to execute the statements following the selected case until the end of the switch statement.

You're going to learn how to exit from the switch construct in the next section.

**The break Statement**

You can add a break statement at the end of the statement list following every case label, if you want to exit the switch construct after the statements within a selected case are executed.

The program in Listing 10.5 does a similar job as the one in Listing 10.4, but this time, the break statement is used.

**TYPE**

**Listing 10.5. Adding the break statement.**

1: /\* 10L05.c Adding the break statement \*/

2: #include <stdio.h>

3:

4: main()

5: {

6: int day;

7:

8: printf("Please enter a single digit for a day\n");

9: printf("(within the range of 1 to 7):\n");

10: day = getchar();

11: switch (day){

12: case `1':

13: printf("Day 1 is Sunday.\n");

14: break;

15: case `2':

16: printf("Day 2 is Monday.\n");

17: break;

18: case `3':

19: printf("Day 3 is Tuesday.\n");

20: break;

21: case `4':

22: printf("Day 4 is Wednesday.\n");

23: break;

24: case `5':

25: printf("Day 5 is Thursday.\n");

26: break;

27: case `6':

28: printf("Day 6 is Friday.\n");

29: break;

30: case `7':

31: printf("Day 7 is Saturday.\n");

32: break;

33: default:

34: printf("The digit is not within the range of 1 to 7.\n");

35: break;

36: }

37: return 0;

38: }

**OUTPUT**

With the help from the break statement, I can run the executable file 10L05.exe and only obtain the output of the selected case:

C:\app>10L05

Please enter a single digit for a day

(within the range of 1 to 7):

1

Day 1 is Sunday.

C:\app>

**ANALYSIS**

This program has seven case labels followed by the constant expressions of `1', `2', `3', `4', `5', `6', and `7', respectively. (See lines 12, 15, 18, 21, 24, 27, and 30.) In each case, there is a statement followed by a break statement. As mentioned, the break statements help to exit the

switch construct after the statement in a selected case is executed. For example, after the int variable day is assigned the value of 1 and evaluated in the switch statement, the case with

`1' is selected, and the statement in line 13 is executed. Then, the break statement in line 14 is executed, which breaks the control of the switch statement and returns the control to the next statement outside the switch construct. In Listing 10.5, the next statement is the return statement in line 37, which ends the main function. The printf() function in line 13 outputs a string of Day 1 is Sunday. on the screen. Note that in a switch statement, braces are not needed to group the statements within an individual case into a statement block.

You can also use the break statement to break an infinite loop. As you saw in Hour 7, the following for and while loops are all infinite loops.

**The continue Statement**

Instead of breaking a loop, there are times when you want to stay in a loop but skip over some statements within the loop. To do this, you can use the continue statement provided by C. The continue statement causes execution to jump to the top of the loop immediately.

You should be aware that using the continue statement, as well as the break statement, may make your program hard to debug.

For example, Listing 10.7 demonstrates how to use the continue statement in a loop doing sums.

**TYPE**

**Listing 10.7. Using the continue statement.**

1: /\* 10L07.c: Using the continue statement \*/

2: #include <stdio.h>

3:

4: main()

5: {

6: int i, sum;

7:

8: sum = 0;

9: for (i=1; i<8; i++){

10: if ((i==3) || (i==5))

11: continue;

12: sum += i;

13: }

14: printf("The sum of 1, 2, 4, 6, and 7 is: %d\n", sum);

15: return 0;

16: }

**OUTPUT**

After the executable file 10L07.exe is run from a DOS prompt, the output is shown on the screen:

C:\app>10L07

The sum of 1, 2, 4, 6, and 7 is: 20

C:\app>

**ANALYSIS**

In Listing 10.7, we want to calculate the sum of the integer values of 1, 2, 4, 6, and 7. Because the integers are almost consecutive, a for loop is built in lines 9\_13. The statement in line 12 sums all consecutive integers from 1 to 7 (except for 3 and 5, which aren't in the listing and are skipped in the for loop). By doing so, the expression (i==3) || (i==5) is evaluated in the if statement of line 10. If the expression returns 1 (that is, the value of i is equal to either 3 or 5), the continue statement in line 11 is executed, which causes the sum operation in line 12 to be skipped, and another iteration to be started at the beginning of the for loop. In this way, we

obtain the sum of the integer values of 1, 2, 4, 6, and 7, but skip 3 and 5, automatically by using one for loop. After the for loop, the value of sum, 20, is displayed on the screen by the printf() function in the statement of line 14.

**The goto Statement**

I feel that this book would not be not complete without mentioning the goto statement, although I do not recommend that you use the goto statement unless it's absolutely necessary. The main reason that the goto statement is discouraged is because its usage may make the C program unreliable and hard to debug.

The following is the general form of the goto statement:

labelname:

statement1;

statement2;

.

.

.

goto labelname;

Here labelname is a label name that tells the goto statement where to jump. You have to place labelname in two places: One is at the place where the goto statement is going to jump (note that a colon must follow the label name), and the other is the place following the goto keyword. You have to follow the same rules to make a label name as you name a variable or function. Also, the place for the goto statement to jump to can appear either before or after the statement.

QUIZ

1. How many expressions are there in the if statement?
2. Why is the if-else statement an expansion of the if statement?
3. Why do you normally need to add the break statement into the switch statement?
4. What can the continue statement do inside a loop?
5. Given x = 0, will the arithmetic operations inside the following if statement be performed?

if (x != 0)

y = 123 / x + 456;

1. Given x = 4, y = 2, and operator = `-', what is the final value of x after the following switch statement is executed?

switch (operator){

case `+': x += y;

case `-': x -= y;

case `\*': x \*= y;

case `/': x /= y;

default: break;

}

7 Similarly to in question 2, using x = 4, y = 2, and operator = `-', what is the final value of after the following switch statement is executed?

switch (operator){

case `+': x += y; break;

case `-': x -= y; break;

case `\*': x \*= y; break;

case `/': x /= y; break;

default: break;

}

1. What is the value of the integer variable x after the following code is executed?

x = 1;

for (i=2; i<10; i++){

if (i%3 == 0)

continue;

x += i;

}

1. ARRAYS

**What Is an Array?**

You now know how to declare a variable with a specified data type, such as char, int, float, or double. In many cases, you have to declare a set of variables that have the same data type. Instead of declaring them individually, C allows you to declare a set of variables of the same data type collectively as an array. An array is a collection of variables that are of the same data type. Each item in an array is called an element. All elements in an array are referenced by the name of the array and are stored in a set of consecutive memory slots.

**Declaring Arrays**

The following is the general form to declare an array:

data-type Array-Name[Array-Size];

Here data-type is the type specifier that indicates what data type the declared array will be. Array-Name is the name of the declared array. Array-Size defines how many elements the array can contain. Note that the brackets ([ and ]) are required in declaring an array. The bracket pair ([ and ]) is also called the array subscript operator. For example, an array of integers is declared in the following statement,

int array\_int[8];

where int specifies the data type of the array whose name is array\_int. The size of the array is 8, which means that the array can store eight elements (that is, integers in this case). In C, you have to declare an array explicitly, as you do for other variables, before you can use it.

**Indexing Arrays**

After you declare an array, you can access each of the elements in the array separately. For instance, the following declaration declares an array of characters:

char day[7];

You can access the elements in the array of day one after another. The important thing to remember is that all arrays in C are indexed starting at 0. In other words, the index to the first

element in an array is 0, not 1. Therefore, the first element in the array of day is day[0]. Because there are 7 elements in the day array, the last element is day[6], not day[7]. The seven elements of the array have the following expressions: day[0], day[1], day[2], day[3], day[4], day[5], and

day[6]. Because these expressions reference the elements in the array, they are sometimes called array element references.

**Initializing Arrays**

With the help of the array element references, you can initialize each element in an array.

For instance, you can initialize the first element in the array of day, which was declared in the last section, like this:

day[0] = `S';

Here the numeric value of S is assigned to the first element of day, day[0]. Likewise, the statement day[1] = `M'; assigns `M' to the second element, day[1], in the array. The second way to initialize an array is to initialize all elements in the array together. For instance, the following

statement initializes an integer array, arInteger:

int arInteger[5] = {100, 8, 3, 365, 16};

Here the integers inside the braces ({ and }) are assigned to the corresponding elements of the array arInteger. That is, 100 is given to the first element (arInteger[0]), 8 to the second element (arInteger[1]), 3 to the third (arInteger[2]), and so on.

Listing 12.1 gives another example of initializing arrays.

**TYPE**

**Listing 12.1. Initializing an array.**

1: /\* 12L01.c: Initializing an array \*/

2: #include <stdio.h>

3:

4: main()

5: {

6: int i;

7: int list\_int[10];

8:

9: for (i=0; i<10; i++){

10: list\_int[i] = i + 1;

11: printf( "list\_int[%d] is initialized with %d.\n", i, list\_int[i]);

12: }

13: return 0;

14: }

**OUTPUT**

The following output is displayed on the screen after the executable (12L01.exe) of the program in Listing 12.1 is

created and run from a DOS prompt:

C:\app>12L01

list\_int[0] is initialized with 1.

list\_int[1] is initialized with 2.

list\_int[2] is initialized with 3.

list\_int[3] is initialized with 4.

list\_int[4] is initialized with 5.

list\_int[5] is initialized with 6.

list\_int[6] is initialized with 7.

list\_int[7] is initialized with 8.

list\_int[8] is initialized with 9.

list\_int[9] is initialized with 10.

C:\app>

**ANALYSIS**

As you can see in Listing 12.1, there is an integer array, called list\_int, which is declared in line 7. The array list\_int can contain 10 elements.

Lines 9\_12 make up a for loop that iterates 10 times. The statement in line 10 initializes list\_int[i], the ith element of the array list\_int, with the value returned from the i + 1 expression.

Line 11 then prints out the name of the element, list\_int[i], and the value assigned to the element.

**The Size of an Array**

As mentioned earlier in this lesson, an array consists of consecutive memory locations. Given an array, like this:

data-type Array-Name[Array-Size];

you can then calculate the total bytes of the array by the following expression:

sizeof(data-type) \* Array-Size

Here data-type is the data type of the array; Array-Size specifies the total number of elements the array can take. The result returned by the expression is the total number of bytes the array takes.

Another way to calculate the total bytes of an array is simpler; it uses the following expression:

sizeof(Array-Name)

Here Array-Name is the name of the array.