Lesson 2

RELATIONAL, LOGICAL, AND EQUALITY OPERATORS

Statements in a program are normally executed one after another. This is called sequential flow of control. Often it is desirable to alter the sequential flow of control to provide for a choice of action, or a repetition of action. By means of if and if-else statements a selection among alternative actions can be made. By means of while, for, and do statements iterative actions can be taken. These flow of control constructs are explained in this lesson.

Since the relational, equality, and logical operators are heavily used in flow of control constructs, the lesson begin with a thorough discussion of these operators. These operators are used in expressions that the programmer thinks of as true or false. The lesson explains how true and false are implemented in C. The lesson also discusses the compound statement. The double { } are used to group together statements that are treated as a unit. In previous lessons the "less than or equal" operator in the expression

```
k <= 5
```

was used to control a while loop. This is an example of a relational expression. The natural interpretation is that this expression is true when the value of k is less than or equal to 5, and false when the value of k is larger than 5. The relational, equality and logical operators are used in expressions in flow of control constructs that control the order in which statements in a program are executed. These expressions have either the value 0 or the value 1. The reason for this is that in C the value zero is used to represent false and any nonzero value is used to represent true. This lesson will carefully explain how these operators work, how they combine with other operators to make expressions, and how they are used in flow of control constructs. The names of the relational, equality, and logical operators, along with the symbols that represent them, are given in the following table.

Type	Operator	Name
relational	<	less than
	<=	less than or equal
	>	greater than

University of Houston Clear Lake

Type	Operator	Name
	>=	greater than or equal
equality	==	equal
	!=	not equal
logical	&&	and
	||	or
	!	not

As with all operators in C, the relational, equality, and logical operators have rules of precedence and associativity that determine precisely how expressions involving these operators are evaluated. All the relational, equality, and logical operators are binary, except for not ! , which is unary. Notice that the precedence of ! is the same as all the other unary operators. The precedence of all the relational, equality, and logical operators other than ! is below the arithmetic operators, but above the assignment operators. Intuitively, an expression such as a < b is either true or false. C evaluates true expressions to the int value 1 and false expressions to the int value 0. As we shall see, any nonzero value is considered true. The relational operators <, < =, > , and > == are all binary. They each take two expressions as operands and yield either the int value 0 or the int value 1 . Some examples are

```
a < 3

a > b + c

-13.7 >= (20.0 *x + 33.1)
```

but not

```
a =<b /* out of order */

a <=b /* space not allowed */

a >> b /* this is a shift operation */
```

Consider the relational expression a < b. If the value of a is less than the value of b, then the expression has the int value 1, which we think of as true. If the value of a is not less than the value of b, then the expression has the int value 0, which we think of as false. Observe that the value of a < b is the same as the

value of a -b< 0. Because the precedence of the relational operators is less than that of the arithmetic operators, the expression

a -b < 0 is equivalent to (a - b)

The equality operators == and != are binary operators acting on expressions. They yield either the int value 0 or the int value 1. Some examples are

```
c == 'A'

k!= -2

x+y==2*2-3
```

but not

```
a = b          /* an assignment statement */

a=  =b - 1     /* space not allowed */

x = !44        /* equivalent to x = (!44) */

x+y = !44      /* equivalent to (x+y) = (!44) - syntax error */
```

Intuitively, an equality expression such as a == b is either true or false. An equivalent expression is a - b = = 0. If the value of a equals the value of b, then a - b has value 0 and 0 == 0 is true. In this case the expression a == b will yield the int value 1. If the value of a is not equal to the value of b, then a == b will yield the int value 0. The expression a != b makes use of the "not equal" operator. It is evaluated in a similar fashion, except that the test here is for inequality rather than for equality.

LOGICAL OPERATORS AND EXPRESSIONS,

The three logical operators are ! , &&, and II. Of these, ! is unary, whereas && and II are binary. All of these operators when applied to expressions yield either the int value 0 or the int value 1.

Logical negation can be applied to an arbitrary arithmetic expression. If an expression has value zero, then its negation will yield the int value 1. On the other hand, if the expression has a nonzero value, then its negation will yield the int value 0. Some examples are

```
!5

!a
```

```
! (x + 8.7)
```

but not

```
a != b/* this is the "not equal" operator */
```

While logical negation is a very simple operator, there is one subtlety. This operator is unlike the not operator in ordinary logic. If s is a logical statement, then

```
not(not s) = s
```

whereas in C the value of ! !5, for example, is 1. Since ! associates from right to left,

```
! !5 is equivalent to          !(!5)
```

and !(!5) is equivalent to !(0), which has the value 1.

The binary logical operators && and II also act on expressions and yield either the int value 0 or the int value 1. Some examples are

```
2 II 3

a && b

b II a == c

x = 2.2 && x <= 8.7 * y
```

but not

```
a &&          /*   one operand missing */

a I b         /*   this is a bitwise operation */

a I  I b      /*   extra space not allowed */

&a  /*        /*   address of a */

a & b         /*   this is a bitwise operation */
```

The precedence of && is higher than II, but both operators are of lower precedence than all unary, arithmetic, and relational operators. Their associativity is "left to right." In the evaluation of expressions that are the operands of && and II, the evaluation process is from left to right, and the process stops as soon as the outcome true or false is known. Suppose that exp1 and exp2 are expressions and that exp1 has value zero. In the evaluation of the logical expression

```
exp1 && exp2
```

the evaluation of exp2 will not occur, because the value of the logical expression as a whole is already determined to be 0. Similarly, if exp1 has nonzero value, then in the evaluation of

```
exp1 II exp2
```

the evaluation of exp2 will not occur, because the value of the logical expression as a whole is already determined to be 1. The following code illustrates this.

```
int i, j = 2;

i = 0 && (j=3);

printf("%d %d\n", i, j);        /* 0 2 is printed */

i =  4  II  ( j=0 );

printf("%d %d\n", i, j);        /* 1 2 is printed */

i  =  0  II  ( j=6 );

printf("%d %d\n", i, j);        /* 1 6 is printed */
```

COMPOUND STATEMENTS

A compound statement is a series of statements surrounded by the braces { and }. The use of the compound statement is to group statements into an executable unit. When declarations come at the beginning of a compound statement, it is called a block. In C wherever it is possible to place a statement, it is also possible to place a compound statement. An example of a compound statement is:

```
{
        ++j;
        sum += x:
```

```
        printf("Index value is : %d\nRunning sum is : %d\n", j, sum)
}
```

Grouping of statements is used to achieve the desired flow of control in such constructs as the if statement and the while statement.

THE NULL STATEMENT

The null statement is written as a single semicolon. It causes no action to take place. An example is

```
a=b+c;

;               /* semicolon this is a null statement */
```

This is not a typical example of the use of a null statement, because here it serves no useful purpose. Usually a null statement is used where a statement is required syntactically, but no action is desired. This situation sometimes occurs in statements that affect the flow of control. An example of a while statement making use of a null statement occurs in Lesson 6.

THE if STATEMENT

The general form of an if statement is

```
if (expression)

statement
```

If expression is nonzero (true), then statement is executed otherwise statement is skipped. After the if statement has been executed, control passes to the next statement. In the example

```
if (grade  >=  90)

   printf("congratulations!\n");

printf("Your grade is %d\n", grade);
```

a congratulatory message is printed only when grade is greater than or equal to 90. The second printf( ) statement is always executed. Usually the expression in an if statement is a relational or equality or logical expression, but an expression from any domain is permissible. Some other examples of if

statements are

```
If (z ! =  0)
x = y/z;


if(c' '){
  ++ blank_cnt;
  printf ("found another blank\n"):
}
```

but not

```
if b  ==  a      /* missing parentheses */


area = a* a;
```

Where appropriate, compound statements should be used to group a series of statements under the control of a single if expression. The following code consists of two if statements.

```
if (j  <  k)

  min = j;

if (j  <  k)


printf("j is smaller than k\n");
```

The code can be written more efficiently and more understandably by making use of a single if statement with its body consisting of a compound statement.

```
if(j < k)  {

  min = j;

  printf("j is smaller than k\n");

}
```

In the first version, the relational expression j < k is evaluated twice, whereas in the second version it is evaluated only once with a compound statement grouping all of the actions needed when the expression is true.

Closely related to the if statement is the if-else statement. It has the general form

```
if (expression)

  statement1

else
  statement2
```

If expression is nonzero, then statement1 is executed and statement2 is skipped; if expression is zero, then statement1 is skipped and statement2 is executed. After the if-else statement has been executed, control passes to the next statement. Consider the code

```
if (x  <  y)

  min = x;

else

  min = y;

printf ("min = %d\n", min);
```

If x < y is true, then min will be assigned the value of x, and if it is false then min will be assigned the value of y. After the if-else statement is executed, the printf( ) statement is executed, causing the value of min to be printed. Another example of an if-else statement is

```
if ( 'a'  <=  c && c  <=  'z')  {

   ++ lower_cnt;

   printf ("Found another lowercase letter.\n");

}

else {

   + + upper_cnt;
   printf("The letter %c is not a lowercase letter.\n", c);
}
```

but not

```
if ( i !=  j )  {
```

```
    i = 5;

    j = 6;

};              /* syntax error */
else

i += j;
```

The syntax error occurs because the semicolon following the right brace creates a null statement, leaving the else with no if attachment. Since an if statement is itself a statement, it can be used as the statement part of another if statement. Consider the code

```
if  (a == 7)
   if (b  ==  7)
printf ("Both a and b are equal to 7");
```

This is of the form

```
if (a  ==  7)

statement
```

where "statement" is the following if statement

```
if (b  ==  7)
printf ("Both a and b are equal to 7");
```

In a similar fashion an if-else statement can be used as the statement part of another if statement. Consider the example

```
if (z == 0)
   if (x == 1)
   printf("z is zero and x is equal to one.");
else
   printf ("z is not zero or x is not equal to one?");
```

The rule is: An else attaches to the nearest unattached if. Thus the code is correctly formatted as the form

```
if (a  ==  1)
statement
```

where statement is the if-else statement

```
if (x == 7)

    printf ("Your in the $$$");

else

   printf("Try again.");
```

To illustrate the use of the if and if-else statements, an interactive program is given that will find the minimum of three values entered at the keyboard.

```
/* Find the minimum of three integers. */
main (   )
{
  int  x,  y,  z,  min;
  printf ("Enter three integers:      \n");
  printf (">>");
  scanf ("%d%d%d", &x, &y, &z);
  if (x < y)
    min = x;
  else
    min = y;
  if (z < min)
    min = z;
  printf ("The minimum value of the three integers is %d\n", min);
}
```

PROGRAM ANALYSIS

```
printf ("Enter three integers:      \n");
  printf (">>");
```

In an interactive environment, the program must prompt the user for input data.

```
scanf ("%d%d%d", &x, &y, &z);
```

The input function scanf ( ), from the standard library stdio.h, is used to read in three integer values that are stored at the address & of x, the address of y, and the address of z, respectively. The address operator & is used for this purpose.

```
if (x   <y)
```

```
  min = x;
else
   min = y;
```

This is a single if-else statement. The values of x and y are compared. If x is less than y, then min is assigned the value of x; if x is not less than y, then min is assigned the value of y.

```
if (z  <  min)
min = z;
```

This is another single if statement. A test is made to see if the value of z is less than the value of min. If it is, then min is assigned the value of z. Otherwise the value of min is left unchanged.

Just as an if-else statement can be used as the statement part of another if statement, so can it be used as the statement part following an else. Here is an example of this.

```
if (a == 1)

   printf("a is equal to 1\n");

else if (a = = 2)

   printf("a is equal to 2\n");

else

   printf("a is not equal to 1 or 2\n");
```

Because long chains of if-else statements can occur, the format convention is as shown above, rather than like the following:

```
if (a  ==  1)

   printfC'a is equal to 1\n");

else

      if (a  ==  2)
         printf("a is equal to 2\n");
      else
         printf("a is not equal to 1 or 2\n");
```

The last example shows the flow of control more clearly, however it is not the preferred C formatting style.

THE while STATEMENT

Repetition of action was one reason why computers were invented. When there are large amounts of data to be process repetitively, it is very convenient to have a control mechanisms to repeat execution on the specific statements. One basic iterative statement is the while statement. The general form of a while statement, or while loop is:

```
while (expresion)

   statement
```

The C compiler first evaluates the expression. If it is nonzero (true), then statement is executed and control passes back to the beginning of the while loop. Notice that the value is nonzero and not necessarily one. The effect of this is that the body of the while loop, namely statement, is executed repeatedly until expression is zero (false). At that point control passes to the next statement. The effect of this is that statement can be executed zero or more times. An example of a while statement is:

```
while (k  <= 10) {
  sum  + =  k;
  ++k;
}
```

Assume that just before this loop the value of k is 1 and the value of sum is 0. Then the effect of the loop is to repeatedly increment the value of sum by the current value of k and then to increment k by 1. After the first time through the loop the value of sum is 0 + 1, after the second time through the loop the value of sum is 0 + 1 + 2, after the third time through the loop the value of sum is 0 + 1 + 2 + 3, and so forth. After the body of the loop has been executed 10 times, the value of k is 11 and the value of the expression k <= 10 is 0 (false). Thus the body of the loop is not executed and control passes to the next statement. When the while loop is exited, the value of sum is 55. The compound statement in brackets is used to group statements together. The compound statement in brackets syntactically representing a single statement.

EXAMPLE

A common task is to find an item having a particular property (max or min) from a

collection of data. The task is to find the maximum value of a real number entered interactively from the keyboard. The program makes use of the if and while statements.

```
/* Find the maximum of n real values. */
main ( )
{
  int   i, n;
  float   max, x;
  printf ("The maximum value will be computed.\n") ;
  printf ("How many numbers do you wish to enter?\n");
  printf (">> ");
  scant ("%d", &n);
  while (n <= 0) {
     printf ("\nERROR: A positive integer is required, \n\n");
     printf ("How many numbers do you wish to enter?\n");
     printf (">> ");
     scanf("%d", &n);
  }/* end while */
  printf ("\nNow enter %d real numbers:\n", n);
 printf (">> ");
  scant ("%f", &x);
  max = x;
  i=1;
  while (i < n) {
     scanf("%f", &x);
     if (max <  x)
         max = x;
     ++i;
  }
  printf ("\nThe maximum value found is =   %g\n",max);
}/* end main program*/
```

If the program is compiled and executed with, 5 when prompted, and then the numbers 2.01, -7, 1.1, 8.07000, and 6. The results on the screen are:

```
The maximum value will be computed.
How many numbers do you wish to enter?
>> 5
Enter 5 real numbers:
>> 2.01 -7 1.1 8.07000 6

Maximum value:  8.07
```

PROGRAM ANALYSIS

```
int i, n;
float max, x;
```

The variables i and n are declared to be of type int, and the variables max and x are declared to be of type float.

```
printf ("The maximum value will be computed.\n"),
```

This line of text is printed to explain the purpose of the program. This is a documentation aid and the program is self-documenting in its output. This is good programming style for future users of the code.

```
printf ("How many numbers do you wish to enter?\n");
  printf (">> ");

scanf("%d", &n);
```

The user is prompted to input an integer with the >> as a cursor. The function scanf() is used to store the value of the integer entered by the user at the address of n. Again the & is the address operator and is needed.

```
while (n <= 0) {
    printf ("\nERROR: A positive integer is required, \n\n");
    printf ("How many numbers do you wish to enter?\n");
    printf (">> ");
    scanf("%d", &n);
  }/* end while */
```

If n is negative or zero, then the value of the expression n <= 0 is 1 (true), This causes the body of the while loop to be executed. An error message and another prompt is printed, and a new value is stored at the address of n. As long as the value of n is negative or zero, the body of the loop is repeatedly executed. The purpose of this while loop is to provide the program with some input error detection capability. Other input errors, such as typing the letter a instead of a digit, still cause the program to fail. For more robust error detection, we need to look at the actual characters typed by the user. To do this we need character processing tools and strings.

```
printf ("\nNow enter %d real numbers:\n", n);
 printf (">> ");
  scant ("%f", &x);
  max = x;
```

The user is next prompted to input n real numbers. The scanf( ) function uses the format %f to convert the characters in the input stream to a floating point

number and to store the converted value at the address of x. The variable max is assigned the value of x.

```
i = 1:
while (i < n) {
    scanf("%f", &x);
    if (max <  x)
       max = x;
    ++i;
 }
```

The int variable i is assigned the value 1. As long as i is less than n, the while statement is done repeatedly:

1. Read in a real number and store its value at the address of x.

2. Compare the value of x to the value of max. If max is less than x, assign the value of x to max.

3. Increment i by 1.

The body of a while statement is a single statement, in this case a compound statement. The compound statement aids flow of control by grouping several statements together to be executed as a unit.

```
printf ("\nThe maximum value found is =   %g\n",max);
```

The value of max is printed in the %g format. Notice that 8.07000 was entered at the keyboard, but 8.07 was printed. With the %g format, extraneous zeros are not printed.

THE for STATEMENT

The for statement defines a statement or set of statements that are executed repeatedly as directed by the conditions that define a program loop. As an example, the following code makes use of a for statement to sum the integers from 1 to 10.

```
int j, sum = 0;

for (j = 1; j <= 10;++j)

   sum + = j ;
```

Code that is equivalent to this is

```
int j, sum = 0;

j=1;

while (j < = 10) {

   sum + = j;

   ++i;
}
```

The for construction is:

```
for(expression1; expression2; expression3)

   statement next statement
```

is equivalent to

```
expression1;

while (expression2) {

   statement;

   expression3;

}

next statement
```

provided that expression2 is nonempty, and provided that a continue statement is not in the body of the for loop.

First expression1 is evaluated. Typically, expression1 is used to initialize a variable used in the loop. Then expression2 is evaluated. If it is nonzero (true), then statement is executed and expression3 is evaluated and control passes back to the beginning of the for loop again, except that evaluation of expression1 is skipped. Typically, expression2 is a logical expression controlling the iteration or numbr of times through the for loop. This process continues until expression2 is zero (false), at which point control passes to next statement.

Any of the three parts can be omitted, although the semicolons must remain. If expression1 or expression3 is left out, it is simply dropped from the expansion. If

the test, expression2, is not present, it is taken as permanently true, so

```
for (; ;) {

   }
```

is an infinite loop. The flow of control could be broken by a return or break.

The for is clearly superior to the while when there is a simple initialization and reinitialization, since it keeps the loop contro statements close together and visible at the top of the loop. Both the while and for loops share the desirable attribute of testing the termination condition at the top, rather than at the bottom of the loop, as the do-while does.

Some examples of the for loop are:

```
for (i = 1; i <= n; ++i)

factorial * = i;  /** This is i! called i factorial 1*2*3   *n  */


for(j = 1 ; scanf("%d",&value) == 1; + +j) {

     printf ("Count: %d Value is:%5d\n", j, value);

      sum += value);

}
```

but not

```
for (i =0 , i < n, ++i)   /* The semicolons must separate expressions
*/
     sum  += i;
```

Any or all of the expressions in a for statement can be missing, but the two semicolons must remain. If expression1 is missing, then no initialization step is performed as part of the for loop. The code

```
i=1;

sum == 0;

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

    sum += i;
```

will compute the sum of the integers from 1 to 10, and so will the code.

```
i = 1;

sum = 0;

for(  ;  i <= 10  ;)

   sum += i++;
```

The special rule for when expression2 is missing, is that the test is always true. Thus the for loop in the code

```
i = 1;

sum = 0;

for(;;){

   sum +==  i++;

   printf("%d\n", sum);

}/* end of for loop*/
```

is an infinite loop.

THE do-while STATEMENT

The third loop in C, the do-while, tests at the bottom after making each pass through the loop body; the body is always executed at least once. The syntax is

do

```
     statement

while (expression);
```

The statement is executed, then expression is evaluated. If it is true, statement is evaluated again, and so on. If the expression becomes false, the loop terminates. The do-while is much less used than while and for.

An example is

```
do {

   sum += i;

   scanf("%d", &i ):

} while (i >0);
```

Again consider the do-while construction of the form

```
do

   statement

while (expression);

next statement
```

First statement is executed, and then expression is evaluated. If it is nonzero (true), then control passes back to the beginning of the do statement and the process repeats itself. When the value of expression is zero (false), then control passes to next statement.

As an example, suppose you want to read in a positive integer, and you want to insist that the integer is positive.

```
do{

   printf (" Please input a positive integer >>   ");
   scant ("%d", &n);
} while (n <= 0);
```

You will be prompted for a positive integer. A negative or zero value will cause the loop to be executed again, asking for another value. The program will execute until you input a positive integer an then exit the loop after a positive integer has been entered.