RELATIONAL AND LOGICAL OPERATORS

The relational and logical operators, in their order of precedence, are shown in the table below.

Operator	Meaning	Type	Associativity
!	Not	Logical	R-L
>	Greater Than	Relational	L-R
<	Less Than		
<=	Less Than or Equal		
>=	Greater Than or Equal		
= =	Is Equal To	Relational	L-R
!=	Is Not Equal To		
&&	Logical AND	Logical	L-R
||	Logical OR	Logical	L-R
?:	Ternary (Conditional)	Relational	R-L

Relational operators are often used in flow control statements: if, if-else-if constructs, while, do-while, for, and sometimes the switch statement. Action is taken based upon the result of a relational expression. The result of a relational expression is either true or false, where false is zero and true is non-zero.

For example,

```
int x = 3, y = 3, z = 4;

x < y       is false
x < z       is true
x >= y      is true
x >= z      is false
x == y      is true
x == z      is false
```

Logical operators normally take relational expressions as operands. The ! operator, which is a logical operator, takes one operand. Other logical operators take two operands. The result of a relational expression is either true or false.

For example,

```
int x = 3, y = 3, z = 4;

(x < y) || (x < z)      is (false || true)      = true
(x == z) || (x > z)     is (false || false)     = false
(x < y) && (x > z)      is (false && false)     = false
!x && x                 is (false && true)      = false
!(x-3) && (y < z)       is (true  && true)      = true
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

FLOW CONTROL

Program flow is controlled by means of loops, branching, switching, and jumps. The forms for the flow-control statements are shown below. Examples are also provided.

WHILE LOOP – While loops are usually used when the programmer desires to loop a block of C statements an unspecified number of times. The actual number of times the loop iterates is based upon an expression evaluating to a true condition. An expression is immediately evaluated, and if the condition is true, an iteration of the while loop occurs. After the iteration of the while loop, the expression is evaluated again. If the expression evaluates to a true condition, another iteration of the loop occurs; otherwise, the program continues to the next C statement following the while loop. The general form of the while loop is shown below along with an example:

```
while (expression)
{
      C statement #1;
      C statement #2;
      ...
      C statement #N;
}
```

Example,

```
#include <stdio.h>
int main()
{
      int userInput = 0, loopCount = 0;

      while(userInput != -1)
      {
            printf("Loop iteration #%d\n", ++loopCount);

            printf("Enter –1 to exit the while loop: ");
            scanf("%d", &userInput);
      }

      return 0;
}
```

Sample Run,

```
Y:\apc-cc-conduit\Images>a
Loop iteration #1
Enter -1 to exit the while loop: 999
Loop iteration #2
Enter -1 to exit the while loop: 37
Loop iteration #3
Enter -1 to exit the while loop: -5
Loop iteration #4
Enter -1 to exit the while loop: 0
Loop iteration #5
Enter -1 to exit the while loop: -1

Y:\apc-cc-conduit\Images>
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

FOR LOOP – For loops are usually used when the programmer desires to loop a block of C statements a known number of times. Usually, a loop counter is initialized in the initialize section of the for loop. A test condition is evaluated, and if the condition evaluates to a true condition, the C statements in the block are executed. After each iteration of the for loop, an update occurs. Usually, the update will bump up the loop counter by one. The test condition is evaluated, and if true, the process continues. If the condition evaluates to false the program continues to the next C statement following the for loop. The general form of the for loop is shown below along with an example:

```
for (initialize; test condition; update)
{
      C statement #1;
      C statement #2;
      ...
      C statement #N;
}
```

Example,

```
/* Program to demonstrate the for loop */
#include <stdio.h>
int main()
{
      int loopCount;

      for(loopCount = 0; loopCount < 5; loopCount++)
      {
            printf("Loop iteration #%d\n", loopCount);
      }

      return 0;
}
```

Sample Run,

```
Y:\apc-cc-conduit\Images>a
Loop iteration #0
Loop iteration #1
Loop iteration #2
Loop iteration #3
Loop iteration #4

Y:\apc-cc-conduit\Images>
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

DO-WHILE LOOP – Do-while loops are usually used when the programmer desires to loop a block of C statements at least one time and for unspecified number of times thereafter. The loop is guaranteed to iterate at least one time, but the actual number of times the loop iterates thereafter is based upon an expression evaluating to a true condition. The do-while loop iterates one time and then an expression is evaluated. If the expression evaluates to a true condition, another iteration of the loop occurs; otherwise, the program continues to the next C statement following the do-while loop. The general form of the do-while loop is shown below along with an example:

```
do
{
      C statement #1;
      C statement #2;
      ...
      C statement #N;
} while (expression);
```

Example,

```
#include <stdio.h>
int main()
{
      int userInput, loopCount = 0;

      do
      {
            printf("Loop iteration #%d\n", ++loopCount);

            printf("Enter –1 to exit the while loop: ");
            scanf("%d", &userInput);
      } while( userInput != -1 );

      return 0;
}
```

Sample Run,

```
Y:\apc-cc-conduit\Images>a
Loop iteration #1
Enter -1 to exit the while loop: 55
Loop iteration #2
Enter -1 to exit the while loop: 0
Loop iteration #3
Enter -1 to exit the while loop: -222
Loop iteration #4
Enter -1 to exit the while loop: -1

Y:\apc-cc-conduit\Images>
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

IF STATEMENT – The if statement is a branching statement that is used to execute a block of C statements if an expression evaluates to a true condition. If the expression evaluates to a false condition, the program continues to the next C statement following the if statement. The general form of the if statement is shown below along with an example:

```
if (expression)
{
      C statement #1;
      C statement #2;
      ...
      C statement #N;
}
```

Example,

```
#include <stdio.h>
int main()
{
      int userInput;

      do
      {
            printf("Enter a number: ");
            scanf("%d", &userInput);

            if(userInput < 100)
            {
                  printf("The number is less than 100\n");
            }
      } while(userInput < 200);

      return 0;
}
```

Sample Run,

```
Y:\apc-cc-conduit\Images>a
Enter a number: 23
The number is less than 100
Enter a number: -345
The number is less than 100
Enter a number: 100
Enter a number: 159
Enter a number: 17
The number is less than 100
Enter a number: 200

Y:\apc-cc-conduit\Images>
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

IF-ELSE-IF CONSTRUCT – The if-else-if construct is a construct of branching statements that is used to execute one of n blocks of C statements if an expression in the construct evaluates to a true condition. If the if expression and all the subsequent else-if expressions each evaluates to a false condition, the program continues to the next C statement that is located in the follow-through else condition (if it exists) or following the entire if-else-if construct. The general form of the if-else-if construct is shown below along with an example:

```
if (expression #1)
{
      C statement #1;
      C statement #2;
      ...
      C statement #A;
}
else if (expression #2)
{
      C statement #1;
      C statement #2;
      ...
      C statement #B;
}
/* ... */
else if (expression #N)
{
      C statement #1;
      C statement #2;
      ...
      C statement #C;
}
else /* Follow-through else condition (optional) */
{
      C statement #1;
      C statement #2;
      ...
      C statement #D;
}
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

Example,

```
#include <stdio.h>
int main()
{
        int userInput;

        do
        {
                printf("Enter a number: ");
                scanf("%d", &userInput);

                if(userInput < 0)
                {
                        printf("The number is negative\n");
                }
                else if((userInput >= 0) && (userInput < 100))
                {
                        printf("The number is between 0 and 99\n");
                }
                else if((userInput >= 100) && (userInput < 200))
                {
                        printf("The number is between 100 and 199\n");
                }
                else
                {
                        printf("Exiting the program ...\n");
                }
        } while(userInput < 200);

        return 0;
}
```

Sample Run,

```
Y:\apc-cc-conduit\Images>a
Enter a number: -5
The number is negative
Enter a number: 0
The number is between 0 and 99
Enter a number: 99
The number is between 0 and 99
Enter a number: 100
The number is between 100 and 199
Enter a number: 199
The number is between 100 and 199
Enter a number: 200
Exiting the program ...

Y:\apc-cc-conduit\Images>
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

TERNARY OPERATOR – The ternary operator (a.k.a. trinary operator or conditional operator) is used in the place of a simple if-else construct. An expression is evaluated, and one of two expressions is returned based upon the result of the evaluation. It makes the code more compact in most cases although it may not gain anything in terms of programming efficiency.

The general form of the ternary operator is shown below along with an example:

```
expression1 ? expression2 : expression3

Example,
#include <stdio.h>

int main(void)
{
       int a, b;

       printf("Enter two different integers: ");
       scanf("%d%d", &a, &b);

       printf("The bigger integer is: %d\n", (a > b) ? a : b);

       return 0;
}
```

Sample Run,

```
Y:\apc-cc-conduit\Images>a
Enter two different integers: 345 234
The bigger integer is: 345

Y:\apc-cc-conduit\Images>
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

SWITCH STATEMENT – The switch statement is often used in the place of long if-else-if constructs. It is more convenient to use than long if-else-if constructs and it is also more efficient. This is because with the if-else-if construct, a test must be made for each "if" statement, whereas switch blocks generate vector jump tables at compile time so that no test is actually required in the underlying code.

The switch test expression must be an integer value (including type char) and the case labels must be integer constant expressions (that is, expressions that only contain integer constants). You cannot use a variable for a case label. The general form of the switch statement is shown below along with an example:

```
switch (expression)
{
      case constant1:
            C statement #1;
            C statement #2;
            ...
            C statement #A;
      break;

      case constant2:
            C statement #1;
            C statement #2;
            ...
            C statement #B;
      break;
      /* ... */
      case constantN:
            C statement #1;
            C statement #2;
            ...
            C statement #C;
      break;

      default: /* Similar to the follow-through else */
            C statement #1;
            C statement #2;
            ...
            C statement #C;
      break; /* Optional, but good coding practice */
}
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

Example,

```
#include <stdio.h>
int main()
{
	int userInput;

	do
	{
		printf("Enter a number: ");
		scanf("%d", &userInput);

		switch(userInput)
		{
			case 0:
				printf("The number is 0\n");
			break;

			case 1:
				printf("The number is 1\n");
			break;

			case 2:
			case 3:
				if(userInput == 2)
					printf("The number is 2\n");
				else
					printf("The number is 3\n");
			break;

			case 8:
				printf("The number is 8\n");
			break;

			default:
				printf("I don't know what it is...\n");
			break;
		}
	} while(userInput < 200);

	return 0;
}
```

Sample Run,

```
Y:\apc-cc-conduit\Images>a
Enter a number: -5
I don't know what it is...
Enter a number: 0
The number is 0
Enter a number: 1
The number is 1
Enter a number: 2
The number is 2
Enter a number: 3
The number is 3
Enter a number: 4
I don't know what it is...
Enter a number: 8
The number is 8
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

```
Enter a number: 100
I don't know what it is...
Enter a number: 200
I don't know what it is...

Y:\apc-cc-conduit\Images>
```

BREAK AND CONTINUE STATEMENTS

A break statement may be used with the switch statement and/or all the loop statements. The break statement will end the nearest enclosing switch statement or loop statement. When a break statement is encountered in a switch statement, that switch statement ends and the next line of code following that switch statement is executed. When a break statement is encountered in a loop statement, that loop statement ends and the next line of code following that loop statement is executed.

A continue statement may be used with any of the loop statements. The continue statement will end the current loop body iteration of the nearest enclosing loop statement. Another iteration of the loop may occur depending on the test condition that governs the loop iterations.

```
Example,
#include <stdio.h>
#define MAXIMUM 20

int main(void)
{
	int counter, cutOff;

	printf("Enter a positive cut-off number: ");
	scanf("%d", &cutOff);

	printf("\nEVEN VALUES:\n");
	for( counter = 1; counter <= MAXIMUM; counter++)
	{
		if (!(counter % 2)) /* Even number */
			printf("%d ", counter);

		if(counter == cutOff)
			break;
	}

	printf("\nODD VALUES:\n");
	for( counter = 1; counter <= MAXIMUM; counter++)
	{
		if (counter % 2) /* Odd number */
			printf("%d ", counter);

		if(counter == cutOff)
			break;
	}

	return 0;
}
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com

```
Sample Run #1,
Enter a positive cut-off number: 100

EVEN VALUES:
2 4 6 8 10 12 14 16 18 20
ODD VALUES:
1 3 5 7 9 11 13 15 17 19

Sample Run #2,
Enter a positive cut-off number: 15

EVEN VALUES:
2 4 6 8 10 12 14
ODD VALUES:
1 3 5 7 9 11 13 15
```

INSTRUCTOR: Joe Russell
EMAIL: joe_programming@yahoo.com