Flow of control: relational, equality, logical operators and expressions

The statements in a C program are typically executed one after another, a process known as sequential flow of control. However, it is often desirable to alter the sequential flow of control to provide for a choice of action or a repetition of action. For example, suppose we read a real number from the terminal and print its square root. Obviously, we do not calculate the square root of a negative number. Thus, we want to do something else when a negative number is entered. Similarly, suppose that we read a positive integer n and then calculate the sum of integers from 1 to n. The program should stop with an appropriate message if a non-positive integer is entered. Let `s` be the required sum. To calculate `s`, we first initialize `s` to `0` by `s=0`. Then we add `1` to `s` by `s=s+1`, then add 2 to it by `s=s+2` and so on until we execute the statement `s=s+n`. This is an example of a repetition of action.

By means of `if`, `if-else`, and `switch` statements, a selection among alternatives can be made. By means of `whi1e`, `for`, and `do` statements, repetitive actions can be taken. Relational, equality, and logical operators are used in the flow of control constructs. They are used in expressions that we think of as true or false. The table below lists the operators that are mostly used in flow of control constructs.

Relational, equality and logical operators		
Relational operators	less than	`<`
	greater than	`>`
	less than or equal to	`<=`
	greater than or equal to	`>=`
Equality operators	equal to	`==`
	not equal to	`!=`
Logical operators	(unary) negation	`!`
	logical and	`&&`
	logical or	`||`

Similar to other operators, the relational, equality, and logical operators have rules of precedence and associativity that determine how expressions involving these operators are evaluated. We now extend the operator precedence and associativity table discussed in Lecture 6. The following table (see next page) shows the updated operators along with their precedence and associativity. The `!` operator is unary, and the other relational, equality, and logical operators are binary. These new operators act on expressions and yield either `int` value 0 or `int` value 1. In C, *false* is represented by the value zero and *true* is represented by any nonzero value. The value for *false* can be any zero value, such as the integer value `0`, the real value `0.0` or the null character `'\0'` (used in string). For example, the expression `printf("!0=%d !0.0=%d\n",!0,!0.0)` produces the output `!0=1 !0.0=1`. Similarly, the output of the expression `printf("!0.1=%d !1=%d\n",!0.1,!1)` is `!0.1=0 !1=0`.

Category	Operator	Associativity
Postfix operators	`( )` (parenthesis), `++` (postfix increment), `--` (postfix decrement),	left to right
Unary operators	`+` (unary plus), `-` (unary minus), `++` (prefix increment), `--` (prefix decrement), sizeof, (type) (type casting), `!` (logical not)	right to left
Multiplicative operators	`*` (multiplication), `/` (division), `%` (modulus)	left to right
Additive operators	`+` (addition), `-` (subtraction)	left to right
Relational operators	`<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to)	left to right
Equality operators	`==` (equal to), `!=` (not equal to)	left to right
Logical and	`&&`	left to right
Logical or	`||`	left to right
Assignment operators	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	right to left

Relational operators and expressions:

All the relational operators

`<` `>` `<=` `>=`

are binary. They each take two expressions as operands and yield either `int` value `0` or `int` value `1`. Consider the following code segment:

```
int a=4;
double b=3.1,c=4.1;
printf("%d %d %d\n",a<2.5,a<b,a+b>=c); // 0  0  1 is printed
printf("%d %d\n",c-b==1,a<b<c); // 0  1 is printed
```

When a C source file containing this code segment is compiled, then the following warning is generated about the last line:

`warning: comparisons like 'X<=Y<=Z' don't have their mathematical meaning`

Note that the expression `a<b<c` is syntactically correct; however, it may not give the desired output. Due to the associativity of the `<` operator (left to right), the expression `a<b<c` is equivalent to `(a<b)<c`. Since `a<b` is false, `(a<b)` is evaluated to `0` and the resulting expression becomes `0<c` which is true. Thus, the value of the expression `a<b<c` evaluates to `1`, meaning it is true. But mathematically, `a<b<c` is not true. To implement mathematical `a<b<c`, we need to use `&&` (logical and) operator and mathematical `a<b<c` is equivalent to `a<b && b<c`. Note that the usual arithmetic conversions occur in relational expressions. Additionally, we must be cautious about the expressions used in relational operators. For example, the value of `c-b` is equal to `1.0` and as such in `c-b==1`, the left side is `double` 1.0 and the right side `int` 1 is converted to `double` 1.0. Thus, both sides are equal to 1.0, which should make them equal, resulting in 1 (true) in the output. However, we see the output 0 (false). The reason

for the discrepancy lies in the fact that 4.1 and 3.1 are not represented exactly in a finite-bit representation. Hence, the difference is not exactly 1.0, which yields the unexpected output. The other outputs in the above code segment are as expected.

On many machines, an expression `a<b` is implemented as `a-b<0`. Thus, if `expr1` and `expr2` are two arbitrary arithmetic expressions, then the following table shows how the value of `expr1-expr2` determines the values of the relational expressions:

Values of the relational expressions				
`expr1 - expr2`	`expr1 < expr2`	`expr1 > expr2`	`expr1 <= expr2`	`expr1 >= expr2`
positive	0	1	0	1
zero	0	0	1	1
negative	1	0	1	0

Equality operators and expressions:

The equality operators `==` and `!=` are binary operators acting on expressions. They yield either the `int` value `0` or the `int` value `1`. The usual arithmetic conversions are applied to the expressions that are the operands of the equality operators. Consider the following code segment:

```
int i=2,j=3,k=5;
printf("%d %d %d\n",i==j,i+j!=k,-2*i+j-4==-1*k); // 0 0 1 is printed
```

Since the arithmetic operators have higher precedence than equality operators, the expression `i+j!=k` is equivalent to `(i+j)!=k`. Unary operators have higher precedence than arithmetic operators. Thus, the statement `-2*i+j-4==-1*k` is equivalent to `((((-2)*i)+j)-4)==((-1)*k)`. The output of the code segment is obvious.

If `expr1` and `expr2` are two arbitrary arithmetic expressions, then the following table shows how the value of `expr1-expr2` determines the values of the equality expressions:

Values of the equality expressions		
`expr1 - expr2`	`expr1 == expr2`	`expr1 != expr2`
zero	1	0
nonzero	0	1

Note that `a != b` is equivalent to `!(a==b)`. *It is very important to keep in mind the following mistake: writing* `a=b` *instead of* `a==b`. The expression `a=b` is an assignment expression, whereas `a==b` checks the equality of `a` and `b` (here `a` and `b` can be arithmetic expressions). In `a=b`, the value of `b` is assigned to `a` and the whole expression `a=b` assumes the same value. Thus, if `b` is nonzero, then `a=b` is also nonzero, and it becomes true. Similarly, if `b` is zero, then `a=b` is also zero, and it becomes false.

Logical operators and expressions:

The logical operator `!` is unary, and the logical operators `&&` and `||` are binary. All of these, when acting on expressions, yield either `int` value `0` or `int` value `1`. If an expression has value zero, then its negation will yield the `int` value `1`. Similarly, if an expression has a nonzero value, then its negation will yield the `int` value `0`. The following table shows the effects of logical negation `!` on expression:

Values of	
`expr`	`!expr`
zero	1
nonzero	0

Consider the following code segment:

```
int i=2,j=4,m;
m=!i+j*!0;
printf("%d\n",m); // 4 is printed
m=!!i*!-j/3;  // 0 is printed
printf("%d\n",m);
```

Since logical negation `!` has higher priority, the expression `!i+j*!0` is equivalent to `(!i)+(j*(!0))` which is equal to `0+4*1=4`. Unary minus `-` and logical negation `!` have the same priority, and their associativity is from right to left. Thus, `!!i*!-j/3` is equivalent to `((!(!i))*(!(-j)))/3` which is equal to `(1*0)/3=0`.

The effects of logical or `||` and logical and `&&` are shown in the following table.

Values of			
`expr1`	`expr2`	`expr1 || expr2`	`expr1 && expr2`
zero	zero	0	0
zero	nonzero	1	0
nonzero	zero	1	0
nonzero	nonzero	1	1

The precedence of `&&` is higher than `||`, but both operators are lower than all unary, arithmetic, relational, and equality operators. Also, their associativity is from left to right. The usual arithmetic conversions occur in expressions that are the operands of logical operators. Also, in the case of mixed types in an expression, certain values are promoted to match the highest type present in the expression. Consider the following code segment:

```
int i=2,j=3,k=4,m;
double x=1.5,y=2;
m= x && i && !k;
```

```
printf("%d\n",m); // 0 is printed
m=i>j || x<y;
printf("%d\n",m); // 1 is printed
m=i<j && x>y || j-i;
printf("%d\n",m); // 1 is printed
```

The expression `x && i && !k` is equivalent to `(x && i) && (!k)` which is equal to `1 && 0=0`. The second expression `i>j || x<y` is equivalent to `(i>j)||(x<y)` which is equal to `0 || 1 = 1`. Finally, the expression `i<j && x>y || j-i` is equivalent to `((i<j) && (x>y)) || (j-i)` which is equal to `0 || 1=1`

Short-circuit Evaluation: In the evaluation of expressions that are the operands of `&&` and `||`, the evaluation process stops as soon as the outcome, whether true or false, is known. This is called short-circuit evaluation. Suppose that `expr1` and `expr2` are expressions and that `expr1` has value zero. In the evaluation of the logical expression `expr1 && expr2`, evaluation of `expr2` will not occur because the value of the logical expression as a whole is already determined to be `0` independent of the value of `expr2`. Similarly, if `expr1` has nonzero value, then in the logical expression `expr1 || expr2`, evaluation of `expr2` will not occur because the value of the logical expression as a whole is already determined to be `1` independent of value of `expr2`.

To explain further, consider the following code segment:

```
int a,b,c,m;
a=-1,b=2,c=3;
m = ++a || b++ || ++c;  // 0 3 3 1 is printed
printf("%d %d %d %d\n",a,b,c,m);
```

The expression `++a || b++ || ++c` is first grouped as `(++a || b++) || ++c`. Next it is evaluated as `(0 || b++) || ++c`. Now it still can be true, and in the next step it becomes `(0 || 2) || ++c`, which is equal to `1 || ++c`. Now we see that the whole expression is true (irrespective of the value of `++c`) and hence `++c` is not evaluated. Thus, value of `a` and `b` change to `0` and `3` and `m` becomes `1`. However, `c` remains unchanged. On the other hand, if we had the following code segment:

```
int a,b,c,m;
a=-1,b=2,c=3;
m = a++ || b++ || ++c;  // 0 2 3 1 is printed
printf("%d %d %d %d\n",a,b,c,m);
```

then `a++` is evaluated as `-1`, which makes the whole expression true. Hence, `b++` and `++c` are not executed.

Next consider the following code segment, where we replace the `||` by `&&`.

```
int a,b,c,m;
a=-1,b=2,c=3;
m = ++a && b++ && ++c;  // 0 2 3 0 is printed
printf("%d %d %d %d\n",a,b,c,m);
```

Then, `b++` and `++c` are not executed, and the whole expression becomes false. On the other hand, all the expressions are evaluated for the following code segment, and the whole expression becomes true.

```
int a,b,c,m;
a=-1,b=2,c=3;
m = a++ && b++ && ++c;  // 0 3 4 1 is printed
printf("%d %d %d %d\n",a,b,c,m);
```

As a final example, consider the following code segment.

```
int a,b,c,m;
a=-1,b=2,c=3;
m = a++ || b++ && ++c;  // 0 2 3 1 is printed
printf("%d %d %d %d\n",a,b,c,m);
```

Since `&&` has higher priority, the above expression is grouped as `a++ || (b++ && ++c)`. Now `a++` is evaluated as `-1` and hence makes the whole expression true. Thus, the latter expressions are not evaluated.

Return to Lecture Topics