Lecture 4

Operators

Because programming languages such as C solve problems by directly manipulating data, there must be operators to carry out necessary data operations for a particular algorithm. C has many operators and the following will describe assignment operators, arithmetic operators, logical operators and miscellaneous "built-in" operators. **An expression** always has a value and consists of at least one operand and perhaps more than one operand combined with an operator. An operand is what the operator will act on. When operator has 2 operands it is named **binary** operator if it needs only one than it is called **unary** operator.

A statement is a command or instruction (executable statement) carried out by the CPU (central processing unit) of a computer; a program is ultimately a series of statements structured in a way to achieve a goal or solve a problem.

Assignment Operator

The assignment operator = is used in assignment statement for assigning a **value** to a particular **variable.** During assignment, *variable* is considered to be the operand since the assignment operator = will copy the contents of another operand – *value* and store the result in the *variable*. When using the assignment operator = , the assignment statement will always be evaluated from right to left (right associative).

Arithmetic Operators

For **arithmetic expressions**, the fundamental operators are as follows:

+ , - (unary form for positive/negative sign ; and binary form - addition/subtraction)
* (multiplication)
/ (division)
% (modulus or MOD) ---> returns the integer remainder of a division (defined for integers only)

NOTE: operator / will result in an integer value if both operands are integers; otherwise, it will result in an floating point value.

NOTE: operator % is defined for integers only.

Examples:

```
int x = 25;
int y = 7;
printf("%d\n", x / y);      ---> 3
printf("%d\n", x % y);      ---> 4
printf("%d\n", y % x);      ---> 7
```

NOTE: the only possible values for an expression of the form x % y are: 0,1,2,...,y – 1.

Consider the following examples:

Example 1: Suppose a line is 76 inches long. Convert this line in dimensions of feet and inches.

```
feet = 76 / 12;
inches = 76 % 12
```

Example 2: Determine if a given integer is odd or even.

```
if(number % 2 == 0)
 true ---> number is even
 false ---> number is odd
```

Example 3: Determine if a given number is divisble by 3.

```
if(number % 3 == 0)
 true ---> yes
 false ---> no
```

Extended Assignment

The extended assignment involves using the assignment operator = and is used to assign a value to multiple variables. For example, consider:

```
x = y = i = a = 15;
```

The above statement would assign all variables in the statement the value of 15. Remember when using the assignment op, the expression is always evaluated from right to left (right associative), in this case, *a* is assigned 15, *i* is assigned the value of a (15), *y* is assigned the value of *i* (15), and *x* is assigned the value of *y* (15).

Shorthand Assignment Operators

C also has shorthand assignment operators or sometimes called compound assignment operators:

```
n += expression; --> implies n = n + (expression)
n -= expression; --> implies n = n - (expression)
n *= expression; --> implies n = n * (expression)
n /= expression; --> implies n = n / (expression)
n %= expression; --> implies n = n % (expression)
```

Misc. "Built-In" Operators

These are "built-in" operators used for misc. purposes.

1. sizeof(*argument*) --> returns the size in bytes of *argument*, which can be a variable or data type.

2. the increment operator ++ adds 1 to a variable. 3. the decrement operator -- does the opposite and subtracts 1 from a variable.

Example:

```
n = n + 1 is equivalent to n++ and ++n
n = n - 1 is equivalent to n-- and --n
```

n++ and n-- are called post-increments, which mean they are performed after everything else in a statement (including assignment). ++n and --n are called pre-increments, which mean they are performed before anything else in the statement.

Example:

```
int x = 7;
int y = 4;
printf("%d\n", x++ * --y);        --> 21
printf("%d %d\n", x, y);          --> 8    3
n = n + 1    -->    n += 1    -->   n++    -- >    ++n
n = n - 1    -->    n -=  1    -- >  n--      -->      --n
```

Operator precedence (determines which expressions are evaluated before others in a statement) and operator associability are both very important to understand. The best way to understand these concepts is to examine a few examples.

Operator Precedence

General Overview:

```
( ) --> parentheses (high precedence)
unary +, -
*, /, %
binary +, - (low precedence)
```

NOTE: To override the default precedence in a statement, use parentheses ( ) to group expressions that need to be evaluated before others.

Operator Associability

Associability may be left associative (left to right) or right associative (right to left). All arithmetic ops are left associative; assignment ops =, +=, -=, *=, /=, %= are right associative.

Let's examine some examples.

Example 1:

```
int a = 6, b = 5, c = 9;
printf("%d\n", a + b * c);        --> 6 + ( 5 * 9 ) = 51
printf("%d\n", a * b % c);        --> ( 6 * 5 ) % 9 = 3
```

Example 2:

```
a = b = c = d = 6;    --> assigns to a, b, c, and d  a value of 6
```

Example 3:

```
x = y = 5;
a = 8;
b = 5
c = a * b / 3 + a / x;
printf("%d", c);                --> ( 8 * 5 ) / 3 + ( 8 / 5 ) = 13 + 1 = 14

c = 9 + a % b;
printf("%d", c);        --> 12

c = 10 / 3 * ( 2 + b )
printf("%d", c);            --> 10 / 3 * ( 7 ) = 3 * 7 = 21;
```

Logical Expressions

Logical expressions, sometimes called boolean expressions, are expressions that return a value of true or false. A (in C language value 1 or, more generally, any nonzero value) is always associated with a true expression; and value 0 is always associated with a false expression. There will be many times when you as a programmer will have to write code to handle certain (true/ false) situations. If an expression is true, then you will issue some command or execute a sequence of instructions. If an expression is false, then you will issue some other command or instruction. A relational expression is a special type of logical expression in which some type of comparison is performed. The relational operators are >, <, <=, >=, ==, and !=.

>>>>>>>>>
DEFINITION: logical expression - an expression that returns a value of either true (1) or false (0).
<<<<<<<<<
>>>>>>>>>
DEFINITION: relational expression - a special type of logical expression in which some type of comparison is performed.
<<<<<<<<<

Relational Operators

> : greater than
< : less than
>= : greater than or equal to
<= : less than or equal to
= = : is equal to
! = : is not equal to

One of the biggest problems many beginning programmers have is distinguishing between = and == . The first oprtator = is an operator used for assignment purposes; it does not imply equality. The second operator == is a relational operator used to compare two values to see if they are equal; "is equal to". Be careful not to use = in relational expressions when == is needed.

Examples of relational expressions:

(1- a) < (2 * b – 7)
(2- c) ! = -1
(3- b) > c + 4 * 7

A relational expression is always a logical expression. Logical expressions are either relational expressions or a combination of multiple relational expressions joined together by the logical operators: &&, ||, and !.

Logical Operators

&& : logical and
|| : logical or
! : logical not (negation)

Example of a logical expression:

(a < b) || (b < c)

If a = 5, b = 3, and c = 10, the result of this expression is 1 (true).

A quick way to tell if an expression is logical but not relational is if one of the logical operators is being used.

Common Errors:

1) Using = (assignment operator) when == is needed.
2) Using an expression such as:
midTerm || final == 'A' which needs to be changed to midTerm == 'A' || final == 'A'
3) Using an expression such as:
a < b < c which needs to be changed to a < b && b < c
4) Using && when || is needed or vice versa:

In general, the operators have precedence (highest to lowest):

A - arithmetic
R - relational
L - logical

NOTE: The only exception is logical not (!), which is evaluated before arithmetic operators.

Specifically, the overall precedence for operators is:

1 : !, +, - (unary)
2 : *, /, %
3 : +, - (binary)
4 : <, <=, >, >=
5 : ==, !=
6 : &&
7 : ||

Type Conversion (Implicit/Explicit)

When executing assignment statements and performing calculations, data types may be temporarily or permanently converted to other data types through something called type conversion. Implicit conversions take place without the programmer specifically asking for it. Conversions requested by the programmer are called explicit conversions. Implicit conversions cause trouble because the programmer can be unaware of

what is actually happening during execution of the program until the code is tested thoroughly.

Implicit Conversion

>>>>>>>>>

DEFINITION: implicit conversion - type conversion that happens without the programmer specifically asking for it

<<<<<<<<<

It's pretty clear that programmers can run into problems because of implicit conversions. However, if you fully understand data types and how they are defined for certain arithmetic operators, you should be able to avoid them. For instance, if *y* and *z* are float, and *a* is int, when the statement *z = a + y* is executed, the value of *a* will be stored in a temporary memory location as float for use in the rest of the statement. Then, the addition is done (in float form) and finally, the result stored in *z*. During this statement, *a* was implicitly converted into *float* during the execution of the statement.

Another form of implicit conversion may take place during assignment, but not necessarily from a smaller type to a larger type.

Example:

```
int a;
float b = 3.5;
float c = 5.0;
int d = 2;
a = b * c / d;
printf("%d", a);        --> 8
```

What conversion took place in the above example? *d* was implicitly converted to 2.0 during *b * c / d*. The result of that arithmetic is 8.75, but *a* was declared *int*, so *a* was implicitly converted to 8 during the assignment statement.

Explicit Conversion

>>>>>>>>>

DEFINITION: explicit conversion - type conversion requested by the programmer; sometimes called typecasting

<<<<<<<<<

While programmers are unaware of implicit conversions, they are completely in control of explicit conversions. Explicit conversions, or typecasting, can come in handy as you will see in later problems. Typecasting can be easily accomplished and it has a easy to use form which is: **(type) variable** where *type* is whatever data type you choose and *variable* is the variable you are converting or casting. It simply takes the value of variable and converts it into the type specified.

Example 1:

```
double d;
int k;
d = 12.78;
k = ( int )d;        --> k = 12
d = ( double)k;      --> d = 12.0
```

Example 2:

```
int m = 10, n = 4, r;
double q;
q = m / n;        --> q = 2.0
r = ( double )m / n;        --> 2
r = m / n;        --> 2
q = ( double)m / n        --> 2.5
```

Lecturer: Mihail Kulev