ECE 131 – Programming Fundamentals

Module 2, Lecture 4: Data Types, Variables and Expressions – Expressions & Statements

Dr. Daryl Lee

University of New Mexico

C Expressions

- An expression in C can be a variable or constant name, a function name, or a function call. In addition, properly applying a operator to one or more of these expressions also produces an expression. Any valid expression that is enclosed in parentheses is an expression.
- Every C expression evaluates to some value.
- An expression of any type (except void) that identifies a data object is called an lvalue.
- If a value can be assigned to an lvalue, then the lvalue is called a modifiable lvalue.

  E.g., The expression on the left-hand side of the assignment operator "=" must be a modifiable lvalue. This is where the term lvalue comes from.

  Ex:

  ```
      num = 2;
  ```

  `num` must be a modifiable lvalue for this assignment statement to be valid.

C Statements

- A C statement is:
  - any valid expression that is immediately followed by a semicolon, or
  - any one of several special statements that are defined in C (and primarily used to control program flow).
- A group of statements may be collected together by enclosing them in braces. This is known as a compound statement or block.
- Recall that a block may contain variable declarations within it, that these variables will have a scope that is local to the block, and that these variables will override any similarly named variables that reside outside the block.

Arithmetic Expressions

- An arithmetic expression is one that includes arithmetic operators.
- The arithmetic operators in C include:

−	unary minus (negation)
+	unary plus
++	increment
−−	decrement
*	multiplication
/	division
%	modulus
+	addition
−	subtraction

- Operands are variables or expressions which are used by operators to evaluate an arithmetic expression expression.

  Ex: In the expression

  ```
  a + b * c
  ```

  the operators are + and *, while a, b, and c are the operands for these operators.

Arithmetic Expressions

- A unary operator takes a single operand, a binary operator takes two operands, and a ternary operator takes three operands.
- Every arithmetic operator in C (indeed every operator in C) has a precedence and an associativity.
- The precedence rules in C are used to specify which operator is evaluated first when two operators with different precedence are adjacent in an expression.
- The associativity rules in C are used to specify which operator is evaluated first when two operators with the same precedence are adjacent in an expression.
- The precedence and associativity of all the C operators are shown in Table A.5 on pg. 440 of Kochan.

Arithmetic Expressions – Use of Parentheses

- Parentheses in C can be used to form groupings of expressions within a larger expression.
- The expressions contained in parentheses will be evaluated first.
- Parentheses can be nested, i.e., parentheses may enclose other parentheses.
- Parentheses cannot be used to indicate multiplication — the only multiplication operator is ‘$*$’.
- The use of parentheses in C supports the notion that you already have about how parentheses are used in mathematical formula.

  Ex: In the expression `a + b * c`, first `b` and `c` are multiplied together, and then this result is added to `a`; while in `(a + b) * c`, first `a` and `b` are added, and then the result is multiplied by `c`.

Arithmetic Expressions – Precedence & Associativity

Here's a portion of Table A.5:

Operator	Description	Associativity
()	grouping	left to right
−	unary minus (negation)	right to left
+	unary plus	
++	increment	
−−	decrement	
*	multiplication	left to right
/	division	
%	modulus	
+	addition	left to right
−	subtraction	
==	equality	left to right
! =	inequality	
=	assignment	right to left

- Operators grouped together in the table share the same precedence and associativity, and operator groups higher in the table have higher precedence.

Arithmetic Expressions – Precedence & Associativity

What does the expression:

$8 + 5 * 7 \% 2 * 4$

evaluate to?

- There are three operators with the same precedence adjacent in this expression ($*$, %, and $*$). Since these operators associate left-to-right, these operators will be executed in the order:

  $8 + (((5 * 7) \% 2) * 4)$

  $8 + ((35 \% 2) * 4))$

  $8 + (1 * 4)$

  $8 + 4$

- Finally, the lowest precent operator in the expression, '$+$', will be executed, yielding the value of the expression: 12.
- What would the result be if the $*$ and % operators associated right-to-left?

The Assignment Operator

- Most arithmetic expressions involve an assignment.
  Ex: `x = a + b * c;`
  Thus, although it is not an actual arithmetic operator, it is important to understand how the assignment operator works in the context of arithmetic expressions.
- The assignment operator has very low precedence, so all of the operations on the right-hand side of the assignment operator are performed prior to the assignment.
- The left-hand operand supplied to the assignment operator must be an lvalue.
- The associativity of the assignment operator is right-to-left.
  Ex: `x = y = z = 2;`
  will first assign 2 to `z`, then the value that `z` stores will be assigned to `y`, and then the value that `y` stores will be assigned to `x`. I.e., `x`, `y`, and `z` will all be assigned the value 2.
  An equivalent statement: `(x = (y = (z = 2)));`

Other Assignment Operators

- There are actually a host of assignment operators in C that combine some arithmetic operation with assignment.
- Additional assignment operators include:

  $*=,\ /=,\ \%=,\ +=,\ -=,\ \&=,\ \wedge=,\ |=,\ <<=,\ >>=$

  Only the first five of these deal with arithmetic operations, the others deal with bit manipulation, which we'll consider later.

  Ex: The expression

  ```
  a *= b
  ```

  uses only one operator, and is equivalent to,

  ```
  a = a * b
  ```

  which uses two operators.
- These operators are not necessary (i.e., you can always use two operators in place of any one of these equality operators), but they do support the C philosophy of a terse syntax.

Equality Operators

- The equality operator ‘==’ is a binary operator that returns a integer value. The return value is 0 if the two operands are *not* equal, and 1 if they are equal.
- The inequality operator ‘!=’ also returns an integer, with the return values exactly opposite of what was just described, i.e., 1 if the operands are not equal, and 0 if they are.
- Note: In C, the value 0 is taken to mean “false”, while *any* non-zero value is considered “true”.
- Be careful when comparing floating-point values, as rounding may lead to unexpected results. In general, it is better to use comparison operators (discussed later) in this case.

Assignment Operator vs. Equality Operator

One of the most common mistakes in C programming is to use the assignment operator in place of the equality operator.

E.g.,

```
if (x = 2)  {return 1;}
else  {return 0;}
```

rather than,

```
if (x == 2) {return 1;}
else  {return 0;}
```

- In the first case, the value 2 will be assigned to `x`, and the value of the expression tested by the `if` statement will be 2, which is treated as true. Thus, this case will *always* test true, and the value 1 will always be returned.
- The second case is probably what the programmer intended. In this case, a 1 will be returned whenever `x` equals 2, and a 0 will be returned whenever `x` does not equal 2.

Expression Types

- An integer expression is one that contains only integer operands. These expressions always evaluates to an integer value, which can lead to truncation.
  Ex: `5/2` will evaluate to 2 rather than 2.5.
- A floating-point expression is one that contains only floating-point operands. These expressions always evaluates to a floating-point value.
- A mixed-mode expression contains operands that are both integer and floating-point. These expressions must be evaluated operator by operator. If both operands are integers, the result will be an integer; however if one of the operands is floating-point, and the other integer, the result will be a floating-point value.
  Ex: `(5/2 * 5/2.0)` will evaluate to 5, not 6.25.
- The exact rules for how these conversions take place are described in Section 5.17 of Appendix A.