CS11001/CS11002
Programming and Data Structures (PDS)
(Theory: 3-1-0)

Assignments

Assignments

- Initialization during declaration helps one store *constant* values in memory allocated to variables. Later one typically does a sequence of the following:
  - Read the values stored in variables.
  - Do some operations on these values.
  - Store the result back in some variable.
- This three-stage process is effected by an **assignment operation**. A generic assignment operation looks like: *variable* = *expression*;

Assignments are Imperative

- Here *expression* consists of variables and constants combined using arithmetic and logical operators.
- The equality sign (=) is the **assignment operator**.
- To the left of this operator resides the name of a variable.
- All the variables present in *expression* are loaded to the CPU. The ALU then evaluates the expression on these values.
- The final result is stored in the location allocated to *variable*.
- The semicolon at the end is mandatory and denotes that the particular statement is over. It is a statement *delimiter*

Imperative Programming

- A C program typically consists of a sequence of statements. They are executed one-by-one from top to bottom (unless some explicit jump instruction or function call is encountered). This sequential execution of statements gives C a distinctive **imperative** flavor.
- This means that the sequence in which statements are executed decides the final values stored in variables.

Example

```
int x = 43, y = 15; /* Two integer variables are declared and initialized */
x = y + 5; /* The value 15 of y is fetched and added to 5. The sum 20 is stored in the memory location for x. */
y = x; /* The value stored in x, i.e., 20 is fetched and stored back in y. */
```

After these statements are executed both the memory locations for x and y store the integer value 20.

Another example

- Let us now switch the two assignment operations.
- int x = 43, y = 15; /* Two integer variables are declared and initialized */

  y = x; /* The value stored in x, i.e., 43 is fetched and stored back in y. */

  x = y + 5; /* The value 43 of y is fetched and added to 5. The sum 48 is stored in the memory location for x. */

For this sequence, x stores the value 48 and y the value 43, after the two assignment statements are executed.

Assignments with same variables

- The right side of an assignment operation may contain multiple occurrences of the same variable.
- For each such occurrence the same value stored in the variable is substituted.
- Moreover, the variable in the left side of the assignment operator may appear in the right side too.
- In that case, each occurrence in the right side refers to the older (pre-assignment) value of the variable.
- After the expression is evaluated, the value of the variable is updated by the result of the evaluation.

Example

- int x = 5; x = x + (x * x);
- The value 5 stored in x is substituted for each occurrence of x in the right side, i.e., the expression 5 + (5 * 5) is evaluated.
- The result is 30 and is stored back to x.
- Thus, this assignment operation causes the value of x to change from 5 to 30.
- The equality sign in the assignment statement is not a mathematical equality, i.e., the above statement does not refer to the equation $x = x + x^2$ (which happens to have a single root, namely x = 0).

Floating point numbers, characters and array locations may also be used in assignment operations.

```
float a = 2.3456, b = 6.5432, c[5]; /* Declare float variables and arrays */
char d, e[4]; /* Declare character variables and arrays */
 c[0] = a + b; /* c[0] is assigned 2.3456 + 6.5432, i.e., 8.8888 */
c[1] = a - c[0]; /* c[1] is assigned 2.3456 - 8.8888, i.e., -6.5432 */
c[2] = b - c[0]; /* c[2] is assigned 6.5432 - 8.8888, i.e., -2.3456 */
a = c[1] + c[2]; /* a is assigned (-6.5432) + (-2.3456), i.e., -8.8888 */
d = 'A' - 1; /* d is assigned the character ('@') one less than 'A' in the ASCII chart */
e[0] = d + 1; /* e[0] is assigned the character next to '@', i.e., 'A' */
 e[1] = e[0] + 1; /* e[1] is assigned the character next to 'A', i.e., 'B' */
 e[2] = e[0] + 2; /* e[2] is assigned the character second next to 'A', i.e., 'C' */
e[3] = e[2] + 1; /* e[3] is assigned the character next to 'C', i.e., 'D' */
```

Implicit Conversion

- An assignment does an implicit type conversion, if its left side turns out to be of a different data type than the type of the expression evaluated.
- float a = 7.89, b = 3.21; int c; c = a + b;
- Here the right side involves the floating point operation 7.89 + 3.21. The result is the floating point value 11.1. The assignment plans to store this result in an integer variable.
- The value 11.1 is first truncated and subsequently the integer value 11 is stored in c.

Example

```
#include<stdio.h>
main()
{
  float a = -7.89., b = 3;
  int c;
  typedef unsigned long newlong;
  newlong d;
   c = (int) a + b;
   d=c;

   printf("%d\n",c);
   printf("%x\n",d);
}
```

Typecasting Again

- float a = 7.89, b = 3.21;
  int c; c = (int)(a + b);

  The parentheses around the expression a + b implies that the typecasting is to be done after the evaluation of the expression. The following variant has a different effect:

- float a = 7.89, b = 3.21; int c; c = (int)a + b;

- What is the value of c now?

Assignments also return a value.

- int a, b, c; c = (a = 8) + (b = 13);
- Here a is assigned the value 8 and b the value 13. The values (8 and 13) returned by these assignments are then added and the sum 21 is stored in c.
- The assignment of c also returns a value, i.e., 21. Here we have ignored this value.

Assignment is *right associative*

- For example,

  a = b = c = 0;

  is equivalent to a = (b = (c = 0));
- Here c is first assigned the value 0. This value is returned to assign b, i.e., b also gets the value 0. The value returned from this second assignment is then assigned to a. Thus after this statement all of a, b and c are assigned the value 0.

Generation of Expressions

- A constant is an expression.
- A (defined) variable is an expression.
- If E is an expression, then so also is (E).
- If E is an expression and *op* a unary operator defined in C, then *op* E is again an expression.
- If E1 and E2 are expressions and *op* is a binary operator defined in C, then E1 *op* E2 is again an expression.
- If V is a variable and E is an expression, then V = E is also an expression.

--- *These rules do not exhaust all possibilities for generating expressions, but form a handy set to start with.*

Examples

- 53 /* constant */
- -3.21 /* constant */
- 'a' /* constant */
- x /* variable */
- -x[0] /* unary negation on a variable */
- x + 5 /* addition of two subexpressions */
- (x + 5) /* parenthesized expression */
- (x) + (((5))) /* another parenthesized expression */
- y[78] / (x + 5) /* more complex expression */
- y[78] / x + 5 /* another complex expression */
- y / (x = 5) /* expression involving assignment */
- 1 + 32.5 / 'a' /* expression involving different data types */

Non-examples

- 5 3 /* space is not an operator and integer constants may not contain spaces */
- y *+ 5 /* *+ is not a defined operator */
- x (+ 5) /* badly placed parentheses */
- x = 5; /* semi-colons are not allowed in expressions */

Operators in C

Operator	Meaning	Description
-	unary negation	Applicable for integers and real numbers. Does not make enough sense for unsigned operands.
+	(binary) addition	Applicable for integers and real numbers.
-	(binary) subtraction	Applicable for integers and real numbers.
*	(binary) multiplication	Applicable for integers and real numbers.

Operators in C

/	(binary) division	For integers division means "quotient", whereas for real numbers division means "real division". If both the operands are integers, the integer quotient is calculated, whereas if (one or both) the operands are real numbers, real division is carried out.
%	(binary) remainder	Applicable only for integer operands.

Examples

- Here are examples of integer arithmetic:
  - 55 + 21 evaluates to 76.
  - 55 - 21 evaluates to 34.
  - 55 * 21 evaluates to 1155.
  - 55 / 21 evaluates to 2.
  - 55 % 21 evaluates to 13.

  Here are some examples of floating point arithmetic:
  - 55.0 + 21.0 evaluates to 76.0.
  - 55.0 - 21.0 evaluates to 34.0.
  - 55.0 * 21.0 evaluates to 1155.0.
  - 55.0 / 21.0 evaluates to 2.6190476 (approximately).
  - 55.0 % 21.0 is not defined.

  Note: C does not provide a built-in exponentiation operator.

Bitwise Operators

- Bitwise operations apply to unsigned integer operands and work on each individual bit.
- Bitwise operations on signed integers give results that depend on the compiler used, and so are not recommended in good programs.
- The following table summarizes the bitwise operations.
- For illustration we use two unsigned char operands a and b. We assume that a stores the value 237 = (11101101)2 and that b stores the value 174 = (10101110)2.

Operator	Meaning	Example								
&	AND	a = 237	1	1	1	0	1	1	0	1
		b = 174	1	0	1	0	1	1	1	0
		a & b is 172	1	0	1	0	1	1	0	0
|	OR	a = 237	1	1	1	0	1	1	0	1
		b = 174	1	0	1	0	1	1	1	0
		a | b is 239	1	1	1	0	1	1	1	1
^	EXOR	a = 237	1	1	1	0	1	1	0	1
		b = 174	1	0	1	0	1	1	1	0
		a ^ b is 67	0	1	0	0	0	0	1	1
~	Complement	a = 237	1	1	1	0	1	1	0	1
		~a is 18	0	0	0	1	0	0	1	0
>>	Right-shift	a = 237	1	1	1	0	1	1	0	1
		a >> 2 is 59	0	0	1	1	1	0	1	1
<<	Left-shift	b = 174	1	0	1	0	1	1	1	0
		b << 1 is 92	0	1	0	1	1	1	0	0