CS10003: Programming & Data Structures

Dept. of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Autumn 2020

Assignments and Arithmetic Operations

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);
```

What is the value of c?

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 do not need 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.

Examples of Expressions

```
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 <u>not</u> 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	1	1	1	0	1	1	0
<<	Left-shift	b = 174	1	0	1	0	1	1	1	0
		b << 1 is 92	0	1	0	1	1	1	0	0

Multiply by 2 (or powers of 2)

```
#include<stdio.h>
main()
{
 int a;
 int n;
  scanf("%d",&a);
  scanf("%d",&n);
  printf("Result: %d\n",a<<n);
}
```

Divide by 2 (or powers of 2)

```
#include<stdio.h>
main()
{
 int a;
 int n;
  scanf("%d",&a);
  scanf("%d",&n);
  printf("Result: %d\n",a>>n);
}
```

If the number is negative

```
Suppose a=--5, n=1
+5: 0000 0000 0000 0101
        1111  1111  1111  1010
                                 1
     --------------------------------
      1111  1111   1111  1011 >> 1: 1111  1111  1111  1101
What does this represent?
      0000 0000  0000 0010
                                  1
      --------------------------------
      0000 0000 0000 0011 : +3
Therefore, the result is -3 (So, is it integer division ?)
```

Bit Complement Operator

Consider an integer i. How do you make the last 4 bits 0?

Method 1: i = i & 0xfff0;

(requires the knowledge of the size of int)

Method 2: i = (i >> 4)<<4; (requires two shifts)

Method 3: i = i & ~0xf;

Concise Form: i &= ~0xf; (expressions like this when the variable being assigned to and the variable being operated on are same can be written like this).

Extract the nth bit

```
#include<stdio.h>
main()
{
    int i, n;
   int bit;
   scanf("%d",&i);
   scanf("%d",&n);
   bit = (i>>n)&1;
   printf("The %dth bit of %d is %d\n",n,i,bit);
}
```

Problem

Can you use this code (method) to find out the binary representation of an integer value?

Write a C code and check.

Ternary Operator

Consists of two symbols: ? and :

- example,
  - larger = (i > j) : i : j;
- i and j are two test expressions.
- Depending on whether i > j, larger (the variable on the left) is assigned.
  - if (i > j), larger = i
  - else (i,e i<=j), larger = j
- This is the only operator in C which takes three operands.

Comma Operator

```
int i, j;
i=(j=1,j+10);
```

What is the result? j=11.

Operator Precedence and Associativity

An explicitly parenthesized arithmetic (and/or logical) expression clearly indicates the sequence of operations to be performed on its arguments.

However, it is quite common that we do not write all the parentheses in such expressions.

Instead, we use some rules of **precedence** and **associativity**, that make the sequence clear.

For example, the expression

a + b * c conventionally stands for

a + (b * c)

and not for (a + b) * c

Another ambiguity

Let us look at the expression a - b - c

Now the *common* operand b belongs to two same operators (subtraction).

They have the same precedence. Now we can evaluate this as

- (a - b) - c or as
- a - (b - c)
- Again the two expressions may evaluate to different values.
- The convention is that the first interpretation is correct.

In other words, the subtraction operator is *left-associative*.

Associativity and Precedence

Operator(s)	Type	Associativity
++ --	unary	non-associative
- ~	unary	right
* / %	binary	left
+ -	binary	left
<< >>	binary	left
&	binary	left
| ^	binary	left
= += -= *= etc.	binary	right

Unary operators

- Consider ++a and a++
  - there is a subtle difference between the two.
  - Recall that every assignment returns a value.
  - The increment (or decrement) expressions ++a and a++ are also assignment expressions.
- Both stand for "increment the value of a by 1". But then which value of a is returned by this expression? We have the following rules:
  - For a++ the older value of a is returned and then the value of a is incremented. This is why it is called the post-increment operation.
  - For ++a the value of a is first incremented and this new (incremented) value of a is returned. This is why it is called the pre-increment operation.

A sample code

```
#include<stdio.h>
main()
{
  int a, s;
  a=1;
  printf("a++=%d\n",a++);
  printf("++a=%d\n",++a);
}
```

Can lead to ambiguities…

```
#include<stdio.h>
main()
{
  int a, s;
  a=1;
  printf("++a=%d,a++=\n",++a,a++);
  }
```

Conditions and Branching

Think about mathematical definitions like the following. Suppose we want to assign to y the absolute value of an integer (or real number) x. Mathematically, we can express this idea as:

$y=0$ if $x = 0$,

$y = x$ if $x > 0$,

$-x$ if $x < 0$.

Fibonacci numbers

F_n= 0 if n = 0,

F_n = 1 if n = 1,

$F_n = F_{n-1} + F_{n-2}$ if n >= 2.

Conditional World

If your program has to work in such a conditional world, you need two constructs:

- A way to specify conditions (like x < 0, or n >= 2).
- A way to selectively choose different blocks of statements depending on the outcomes of the condition checks.

Logical Conditions

Let us first look at the rendering of logical conditions in C.

A logical condition evaluates to a **Boolean value**, i.e., either "true" or "false".

For example, if the variable x stores the value 15, then the logical condition x > 10 is true, whereas the logical condition x > 100 is false.

Mathematical Relations

Relational operator	Usage	Condition is true iff
==	E_1 == E_2	E_1 and E_2 evaluate to the same value
!=	E_1 != E_2	E_1 and E_2 evaluate to different values
<	$E_1 < E_2$	E_1 evaluates to a value smaller than E_2
<=	E_1 <= E_2	E_1 evaluates to a value smaller than or equal to E_2
>	$E_1 > E_2$	E_1 evaluates to a value larger than E_2
>=	E_1 >= E_2	E_1 evaluates to a value larger than or equal to E_2

Examples

Let x and y be integer variables holding the values 15 and 40 at a certain point in time. At that time, the following truth values hold:

x == y False

x != y True

y % x == 10 True

600 < x * y False

600 <= x * y True

'B' > 'A' True

x / 0.3 == 50 False (due to floating point errors)

A funny thing about C is that it does not support any Boolean data type.

Instead it uses any value (integer, floating point, character, etc.) as a Boolean value.

Any non-zero value of an expression evaluates to "true", and the zero value evaluates to "false". In fact, C allows expressions as logical conditions.

Example:

0	False
1	True
6 - 2 * 3	False
(6 - 2) * 3	True
0.0075	True
0e10	False
'A'	True
'\0'	False
x = 0	False
x = 1	True

The last two examples point out the potential danger of mistakenly writing = in place of ==. Recall that an assignment returns a value, which is the value that is assigned.

Logical Operators

Logical operator	**Syntax**	**True if and only if**
AND	C_1 && C_2	Both C_1 and C_2 are true
OR	C_1 || C_2	Either C_1 or C_2 or both are true
NOT	!C	C is false

Examples

(7*7 < 50) && (50 < 8*8) True

(7*7 < 50) && (8*8 < 50) False

(7*7 < 50) || (8*8 < 50) True

!(8*8 < 50) True

('A' > 'B') || ('a' > 'b') False

('A' > 'B') || ('A' < 'B') True

('A' < 'B') && !('a' > 'b') True

Note

Notice that here is yet another source of logical bug. Using a single & and | in order to denote a logical operator actually means letting the program perform a bit-wise operation and possibly ending up in a logically incorrect answer

Associativity of Logical Operators

Operator(s)	**Type**	**Associativity**
!	Unary	Right
< <= > >=	Binary	Left
== !=	Binary	Left
&&	Binary	Left
||	Binary	Left

Examples

x <= y && y <= z || a >= b is equivalent to

((x <= y) && (y <= z)) || (a >= b).

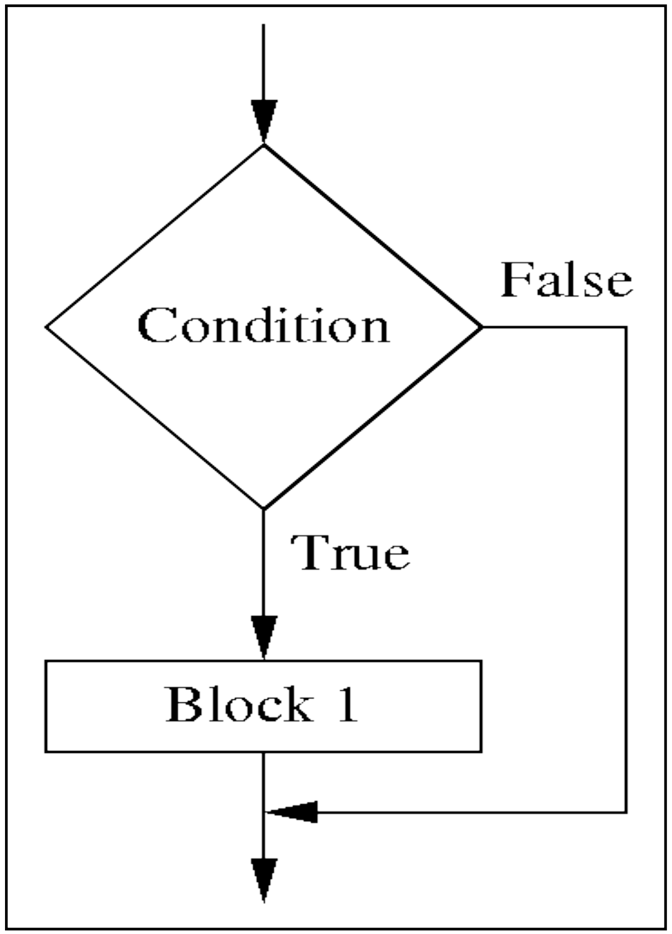
C1 && C2 && C3 is equivalent to

(C1 && C2) && C3.

a > b > c is equivalent to

(a > b) > c.

The If Statement

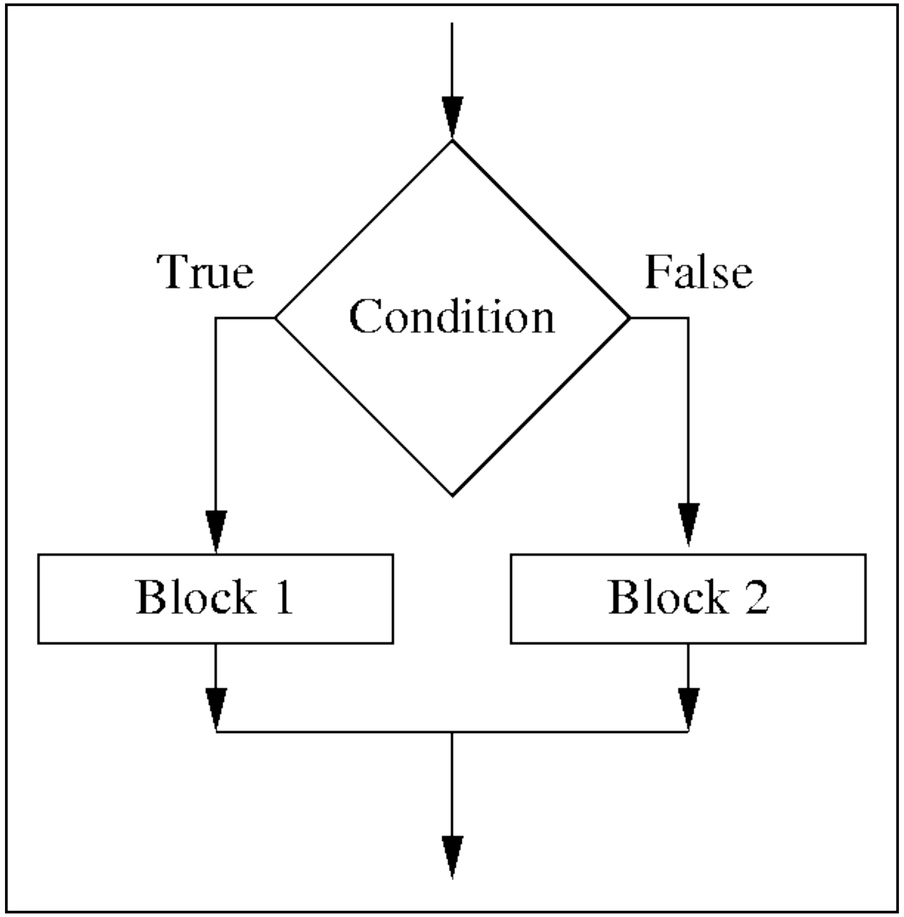


C Statement:

```
   if(Condition)
      Block1;

scanf("%d",&x);
if (x < 0) x = -x;
x=x+1;
```

The If else Statement



<u>C Statement:</u>

```
if (Condition)
 { Block 1 }
else { Block 2 }

scanf("%d",&x);
if (x >= 0) y = x;
else y = -x;
x=x+1;
```

The ternary statement

Consider the following special form of the if-else statement:

if (C) v = E1; else v = E2; Here depending upon the condition C, the variable v is assigned the value of either the expression E1 or the expression E2. This can be alternatively described as:

v = (C) ? E1 : E2; Here is an explicit example. Suppose we want to compute the larger of two numbers x and y and store the result in z. We can write:

z = (x >= y) ? x : y;