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

Link for slides

- **http://cse.iitkgp.ac.in/~debdeep/courses_iitkgp/PDS/index.htm**

Assignments

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

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

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)

Booleaan Values in C

- 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

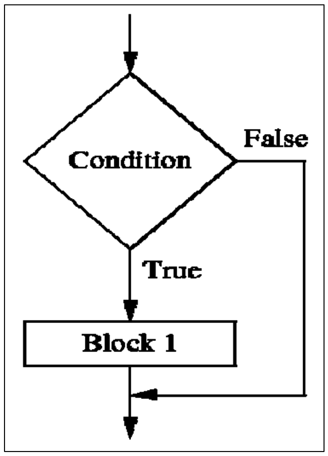
Associativity of Logical Operators

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

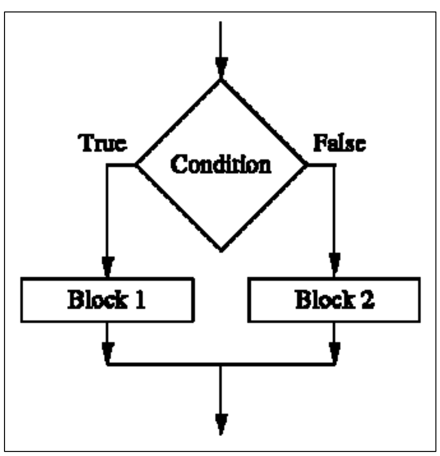


- <u>C Statement:</u>

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;