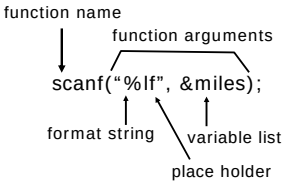
# CS 201
# Selection Structures (1)

Debzani Deb



## Error in slide: scanf Function




- When accepting double values, the placeholder used in `scanf` function is %lf i.e. % then el then ef.



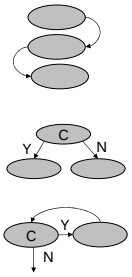
## Overview

- Control Structures
- Conditions
- if statements
- Nested and Multiple if Statements



## Control Structures

- **Control structures** –control the flow of execution in a program or function.
- Three basic control structures:
- **Sequential Flow** - this is written as a group of statements bracketed by { and }where one statement follows another.
- **Selection control structure** - this chooses between multiple statements to execute based on some condition.
- **Repetition –** this structure executes a block of code multiple times.






## Compound Statements

- A **Compound statement** or a Code Block is written as a group of statements bracketed by { and } and is used to specify **sequential** flow.

```
{
   Statement_1;
   Statement_2;
   Statement_3;
}
```

  - Example: the `main` function is surrounded by {}, and its statements are executed sequentially.
  - Function body also uses compound statement.



## Selection in C

- `if` Statement
  - Many form
- `switch` statement
  - Similar to case structure

## Conditions

- A program chooses among alternative statements by testing the values of variables.
  - 0 means false
  - Any non-zero integer means true, Usually, we'll use 1 as true.

```
if (a>=b)
      printf("a is larger");
else
      printf("b is larger");
```

- **Condition -** an expression that establishes a criterion for either executing or skipping a group of statements
  - `a>=b` is a condition that determines which `printf` statement we execute.



## Relational and Equality Operators

- Most conditions that we use to perform comparisons will have one of these forms:
  - variable *relational-operator* variable e.g. a < b
  - variable *relational-operator* constant e.g. a > 3
  - variable *equality-operator* variable e.g. a == b
  - variable *equality-operator* constant e.g. a != 10



## Relational and Equality Operators

| Operator | Meaning | Type |
|---|---|---|
| < | less than | relational |
| > | greater than | relational |
| <= | less than or equal to | relational |
| >= | greater than or equal to | relational |
| == | equal to | equality |
| != | not equal to | equality |



## Logical Operators

- **logical expressions** - expressions that uses conditional statements and logical operators.
  - && (and)
    - A && B is true if and only if both A and B are true
  - || (or)
    - A || B is true if either A or B are true
  - ! (not)
    - !(condition) is true if condition is false, and false if condition is true
    - This is called the **logical complement** or **negation**
- Example
  - `(salary < 10,000) || (dependents > 5)`
  - `(temperature > 90.0) && (humidity > 90)`
  - `!(temperature > 90.0)`



## Truth Table && Operator

| A | B | A && B |
|---|---|---|
| **False** (zero) | **False** (zero) | **False** (zero) |
| **False** (zero) | **True** (non-zero) | **False** (zero) |
| **True** (non-zero) | **False** (zero) | **False** (zero) |
| **True** (non-zero) | **True** (non-zero) | **True** (non-zero) |



## Truth Table || Operator

| A | B | A \|\| B |
|---|---|---|
| **False** (zero) | **False** (zero) | **False** (zero) |
| **False** (zero) | **True** (non-zero) | **True** (non-zero) |
| **True** (non-zero) | **False** (zero) | **True** (non-zero) |
| **True** (non-zero) | **True** (non-zero) | **True** (non-zero) |

## Operator Table ! Operator

| A | !A |
|---|---|
| **False** (zero) | **True** (non-zero) |
| **True** (non-zero) | **False** (zero) |



## Remember!

- && operator yields a true result only when both its operands are true.
- || operator yields a false result only when both its operands are false.



## Operator Precedence

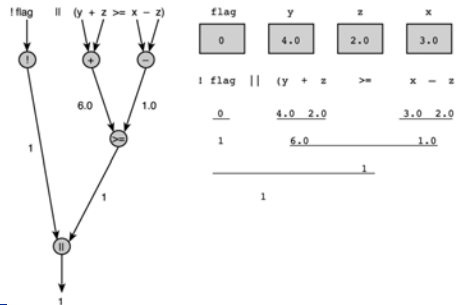
- Operator's precedence determine the order of execution. Use parenthesis to clarify the meaning of expression.
- Relational operator has higher precedence than the logical operators.
- Ex: followings are different.

```
(x<y || x<z) && (x>0.0)
 x<y || x<z && x>0.0
```

function calls
! + - & (unary operations)
*, /, %
+, -
<, >, <=, >=
==, !=
&&
||
=



**Figure 4.1** Evaluation Tree and Step-by-Step Evaluation for !flag || (y + z >= x - z)






## Short Circuit Evaluation (1)

- **Short-circuit (minimal) evaluation** - C stops evaluating a logical expression as soon as its value can be determined.
  - if the first part of the OR expression is true then the overall condition is true, so we don't even look at the second part.
  - if the first part of the AND expression is false then the overall condition is false, so we don't even look at the second part.
  - Example:
    `if ((a<b) && (c>d))` if (a<b) is false, (c>d) is not evaluated.
    `if ((e!=f) || (g<h))` if (e!=f) is true, (g<h) is not evaluated.
  - This can be significant for your program performance.



## Short Circuit Evaluation (2)

```
if (lname == "Smith") && (ssn == 12345678)
if (ssn == 12345678) && (lname == "Smith")
```

- What is the difference?
- AND operator
  - You want to get "false" as soon as possible, since it finishes comparisons
  - i.e. the "most selective" test should be placed at the beginning.
- OR operator
  - You want to get "true" as soon as possible, since it finishes comparisons
  - i.e. the "least selective" test should be placed at the beginning and the "most selective" test at the very end.

## Short Circuit Evaluation (3)

- Short circuit evaluation may cause **problems** for programmers who do not realize (or forget) it is happening.
- Example:
```
if (condition_1 && myfunc(a)) {
    do_something();
}
```
  - If `myfunc(a)` is supposed to perform some required operation regardless of whether `do_something()` is executed, and `condition_1` evaluates as `false`, then `myfunc(a)` will not execute, which could cause problems.
- Don't perform important computations in the second half of a logical expression



## Writing English Conditions in C

- Make sure your C condition is logically equivalent to the English statement.
  - "x and y are greater than z"
    `(x > z) && (y > z)` (valid)
    `x && y > z` (invalid)



## Character Comparison

- C allows character comparison using relational and equality operators.
- During comparison Lexicographic (alphabetical) order is followed. (See Appendix A for a complete list of ASCII values).

```
'9' >= '0'  // True
'a' < 'e'   // True
'a' <= ch && ch <= 'z' /* True if ch is a char type
                          variable that contains a
                          lower case letter.*/
```



## Logical Assignment

- You can assign an `int` type variable to a non zero value for true or zero for false.
```
Ex: even = (n%2 == 0)
    if (even) { do something }
```
- Some people prefer following for better readability.
```
#define FALSE 0
#define TRUE !FALSE
even = (n%2 == 0)
    if (even == TRUE) { do something }
```
- Some people define a separate variable type `boolean`.
  - We'll see that in enumerated types in Ch 7.



## Complementing a condition

- We can complement a logical expression by preceding it with the symbol !.
- We can also complement a single condition by changing its operator.
  - Example : The complement of (`age == 50`) are
    `!(age == 50) , (age != 50)`
  - The relational operator should change as follows
    <= to >, < to >= and so on



## DeMorgan's Theorem (1)

- DeMorgan's theorem gives us a way of simplifying logical expressions.
- The theorem states that the complement of a conjunction is the disjunction of the complements or vice versa. In C, the two theorems are
  1. `!(x || y) == !x && !y`
  2. `!(x && y) == !x || !y`

  Example: If it is not the case that I am tall and thin, then I am either short or fat (or both)
- The theorem can be extended to combinations of more than two terms in the obvious way.

## DeMorgan's Theorem (2)

- DeMorgan's Theorems are extremely useful in simplifying expressions in which an AND/OR of variables is inverted.
- A C programmer may use this to re-write

```
if (!a && !b) ...
```

as

```
if(!(a || b) ...
```

Thus saving one operator per statement.

- Good, optimizing compiler should do the same automatically and allow the programmer to use whatever form seemed clear to them.



## IF() THEN {} ELSE {}

```
if condition
  {compound_statement_1 } // if condition is true
else
  { compound_statement_2 } // if condition is false
```

Example:

```
if (crash_test_rating_index <= MAX_SAFE_CTRI) {
  printf("Car #%d: safe\n", auto_id);
  safe = safe + 1;
}
else {
  printf("Car #%d: unsafe\n", auto_id);
  unsafe = unsafe + 1;
}
```



## IF() THEN {} ELSE {}

- When the symbol { follows a condition or else, the C complier either executes or skips all statements through the matching }
- In the example of the previous slide, if you omit the braces enclosing the compound statements, the if statement would end after the first `printf` call.
- The `safe = safe + 1;` statement would always be executed.
- You MUST use braces if you want to execute a compound statement in an `if` statement.
- To be safe, you may want to always use braces, even if there is only a single statement.



## No {}?

```
if (rest_heart_rate > 56)
  printf("Keep up your exercise program!\n");
else
  printf("Your heart is in excellent health!\n");
```

- If there is only one statement between the {} braces, you can omit the braces.



## One Alternative?

- You can also write the `if` statement with a single alternative that executes only when the condition is true.

```
if ( a <= b )
   statement_1;
```

- Another form – seldom used, but still allowed

```
if ( a == b )
else
   statement_2;
```



## Nested if Statements

- So far we have used if statements to code decisions with one or two alternatives.
- A compound statement may contain more if statements.
- In this section we use nested if statements (one if statement inside another) to code decisions with multiple alternatives.

```
if (x > 0)
  num_pos = num_pos + 1;
else
  if (x < 0)
     num_neg = num_neg + 1;
  else
     num_zero = num_zero + 1;
```

## Comparison of Nested if and Sequences of ifs

- Beginning programmers sometime prefer to use a sequence of if statements rather than a single nested if statement

```
if (x > 0)
   num_pos = num_pos + 1;
if (x < 0)
   num_neg = num_neq + 1;
if (x == 0)
   num_zero = num_zero +1;
```

- The book says this is less readable
  - I disagree with that.
- It is, though, less efficient because all three of the conditions are always tested.
- In the nested if statement, only the first condition is tested when `x` is positive.