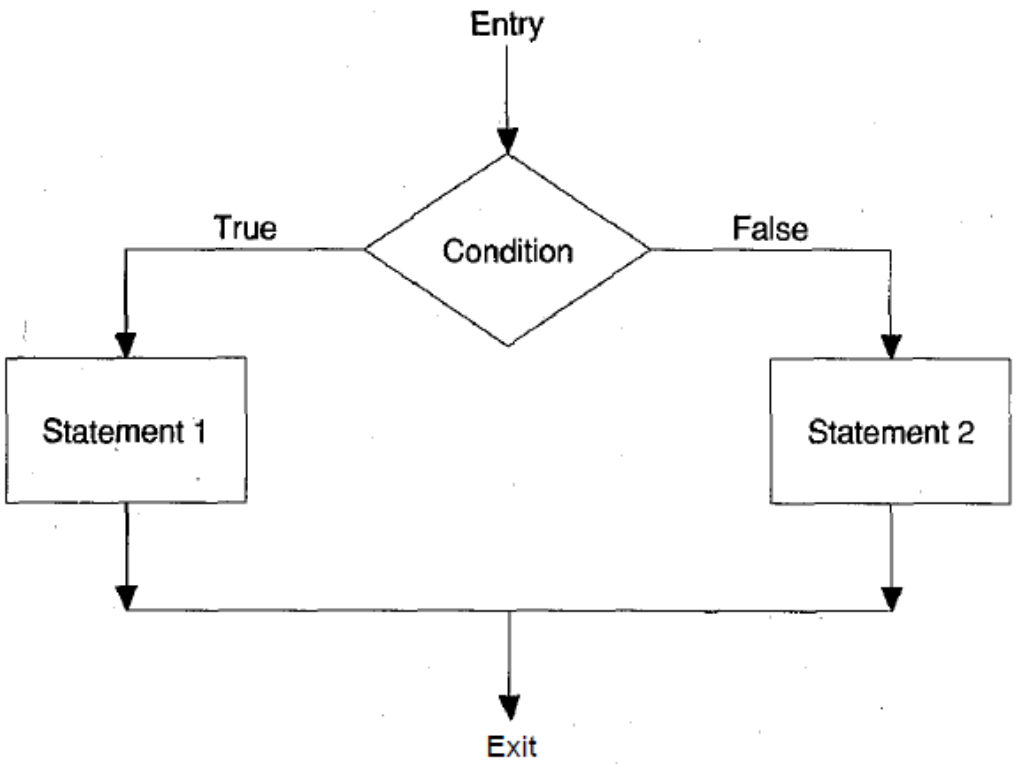
Lecture 2
Selective Structures

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, Kumar

Selective structure



OVERVIEW

Reconsider the program for determining simple interest that we analyzed in Example 2 of Section 2.1. We now assume that the bank offers two interest rates: a bonus rate of 9.75% for those who invest at least $10,000 or keep the principal invested for more than 5 years, and a regular rate of 8.75% otherwise. Here is an algorithm for solving this problem:

1. Read the amount of principal and the number of years for which the principal is to be invested.
2. Determine the interest rate depending upon the principal and the years of investment.
3. Compute the total interest accrued using the formula

   interest = principal x rate x years.

4. Print interest.

```
#include <stdio.h>
#define BONUS 0.0975
#define REGULAR 0.0875

int main(void)
  {
    float principal, rate, interest;
    int     years;

    /* prompt the user to provide input values */
    printf("principal and years of investment? ");

    /* read the input values */
    scanf("%f %d", &principal, Syears);

    /* determine the interest rate */
    if (principal >= 10000 | | years > 5)
        rate = BONUS;
    else
        rate = REGULAR;

    /* compute interest */
    interest = principal * rate * years;

    /* print interest */
    printf("interest = %f\n", interest);

    /* successful completion */
    return 0;
  }
```

The expression enclosed within parentheses following the keyword **if:**

```
principal >= 10000 | | years > 5
```

is the test that determines whether **rate** is set to **BONUS** or **REGULAR.** If the condition is satisfied, i.e., if the expression is *true,* then the statement controlled by **if:**

```
rate = BONUS;
```

is executed; if the condition is not satisfied, i.e., the expression is *false,* then the statement controlled by **e l s e:**

```
rate = REGULAR;
```

is executed. Thus, the program follows one of the two alternative paths depending upon the result of the evaluation of the test condition during the processing of this selective structure.

C does not have Boolean values: *true* and *false*. Instead, integers are used as substitutes for Boolean values. Any nonzero value is interpreted as *true* and zero is interpreted as *false*.

The test expression

```
principal >= 10000. I | years > 5
```

consists of two *relational expressions*

```
principal >= 10000
```

and

```
years > 5
```

combined with the *logical OR operator*

I I

into a *logical expression.*

Logical operators are used for combining expressions, usually relational expressions, into logical expressions. The three logical operators are: I I (logical OR), & & (logical AND), and ! (logical NOT).

More than one action can be specified within an alternative path in a selective structure by grouping statements into a *block*, also called a *compound statement*. A *block* is a sequence of variable declarations and statements enclosed within braces. Moreover, it is not necessary to have an **`el se`** part in an **`i f`** statement. Thus, if we wanted to ensure that the values provided by the user for **`principal`** and **`years`** were nonnegative, we could add the statement

```
if (principal < 0 || years < 0)
   {
     printf("negative values in input\n");
     return 1;
   }
```

C also provides a *conditional expression operator* that is often used to assign one of the two values to a variable depending upon some condition. Using this operator, **rate** can be set to either **BONUS** or **REGULAR** as follows:

```
rate = principal >= 10000 || years > 5 ? BONUS : REGULAR;
```

RELATIONAL OPERATORS

C provides six relational operators for comparing the values of two expressions, and the expression so formed is called a *relational expression.*

Relational operators can be applied to operands of any arithmetic type. The result of comparison of two expressions is *true* if the condition being tested is satisfied and *false* otherwise.

However, C has no special data type for logically valued quantities.

The value of a relational expression, instead, is of type **`int`**,

and is 1 if the result of comparison is true and **`0`** otherwise. For example,

`15 > 10` has the value **`1`** *(true).*
15 <= 10 has the value 0 *(false).*

Relational operators

Relational Operator	Name	Relational Expression
<	Less than	*exp1* < *exp2*
<=	Less than or equal to	*exp1* <= *exp2*
>	Greater than	*exp1* > *exp2*
>=	Greater than or equal to	*exp1* >= *exp2*
==	Equal to	*exp1* == *exp2*
! =	Not equal to	*exp1* ! = *exp2*

A common programming error is to confuse the equal to operator == with the assignment operator =. The expression

```
x == 10
```

tests if the value of `x` is equal to 10, whereas

```
x = 10
```

assigns 10 to the variable x. Thus, the statement

```
if (x == 10) printf("tenbagger");
```

will print `tenbagger` only if `x` is `10`, whereas the statement

```
if (x = 10) printf("tenbagger") ;
```

will print `tenbagger` irrespective of the value of `x` because the value of the assignment expression x = 10 is 10 *(true)*.

Precedence and Associativity

In this table, an operator in a higher row has a higher precedence when compared to an operator in a lower row.
Those operators that are in the same row have the same precedence and associativity.

Precedence and associativity of the relational operators

Operators	Type	Associativity
+ – ++ ––	Unary	Right to left
* / %	Binary	Left to right
+ –	Binary	Left to right
< <= > >=	Binary	Left to right
== ! =	Binary	Left to right
= *= /= %= += -=	Binary	Left to right

```
5  < n < 10
i   == j == 5
p >= q == r >= s
```

interpreted as

```
(5 < n) < 10                  ⇨   1
(i == j) == 5                 ⇨   0
(p >= q) == (r >= s)          ⇨   1  or  0
```

The relational operators have lower precedence than the arithmetic operators but higher than the assignment operator. Thus, the expression

```
a < b + c
```

is interpreted as

```
a < (b + c)
```

However,

```
a = b == c
```

is interpreted as

```
a = (b == c)
```

Finally, the expression

```
a = b + c ! = d + e
```

is interpreted as

```
a = ( (b + c) ! = (d + e) )
```

int i = 3, j = 2, k = 1;

Expression	Equivalent Expression	Value
i > j > k		
i >= j >= k		
i ! = j ! = k		
k < i != k < j		
i - k == j * k		
i > j == i + k > j + k		
i += j != k		
i = k != j < k * j		

int i = 3, j = 2, k = 1;

Expression	Equivalent Expression	Value
i > j > k	(i > j) > k	(3 >2) >1 = 1>1 = 0
i >= j >= k	(i >= j) >= k	(3 >= 2) >= 1 = 1 >= 1 = 1
i ! = j ! = k	(i != j) != k	(3 != 2) != 1 = 1 != 1 = 0
k < i != k < j	(k < i) ! = (k < j)	(1 <3) ! = (1<2) =1 != 1 = 0
i - k == j * k	(i - k) == (j * k)	(3 - 1) == (2 * 1) = 2 == 2 = 1
i > j == i + k > j + k	(i> j) == ((i + k) >(j+k))	1== (4 > 3) = 1 == 1 = 1
i += j != k	i += (j ! = k)	i+=(2!=1) = i+=l = i=4=4
i = k != j < k * j	i=(k! = (j<(k* j)))	i=(l! = (2<2)) = i= (1! = 0) =i=1=1

LOGICAL OPERATORS

C provides three logical operators for combining expressions into *logical expressions*.

The logical operators & & and I I are binary operators, whereas the logical operator ! is a unary operator. The value of a logical expression is 1 or 0, depending upon the logical values of the operands. The operands may be of any arithmetic type. The type of the result is always `int`.

Logical operators

Symbol	Name	Meaning
&&	Logical AND	Conjunction
| |	Logical OR	Disjunction
!	Logical NOT	Negation

Logical AND Operator

The *logical AND* operator combines two expressions into a logical expression and has the following operator formation:

exp1 && *exp2*

An expression of this form is evaluated by first evaluating the left operand. If its value is zero *(false)*, the evaluation is short-circuited and the right operand is not evaluated; the value of the logical expression then is 0 *(false)*. However, if the value of the left operand is nonzero *(true)*, the right operand does get evaluated. The value of the logical expression is 1 *(true)* if the right operand has nonzero *(true)* value, and 0 *(false)* otherwise. Table 3.4 summarizes the results of the

Logical AND operator

expl	exp2	*expl && exp2*
nonzero *{true)*	nonzero *(true)*	l *(true)*
nonzero (true)	*zero (false)*	○ *(false)*
zero *(false)*	nonzero *(true)*	○ *(false)*
zero *(false)*	zero *(false)*	○ *(false)*

```
int a, b, c;
a = b = c = 10;
```

the logical expression

```
a && (b + c)
```

has the value 1 *(true)*, since both a and (b + c) are nonzero *(true)*, whereas the logical expression

```
a && (b - c)
```

has the value 0 *(false)*, since (b - c) is zero *(false)*.

A desirable consequence of the short-circuited left to right evaluation of the && operator is that an expression such as

```
(a != 0) && (b / a > 10)
```

Logical OR Operator

The *logical OR* operator, like the logical AND operator, combines two expressions into a logical expression, and has the following operator formation:

exp1 | | *exp2*

An expression of this form is evaluated by first evaluating the left operand. If the value of the left operand is nonzero *(true)*, the right operand is not evaluated; the value of the logical expression then is 1 *(true)*. However, if the value of the left operand is zero *(false)*, the right operand is evaluated, and the value of the logical expression is 1 *(true)* if the right operand has a nonzero *(true)* value, and 0 *(false)* otherwise.

Logical OR operator

expl	exp2	*expl* || exp2
nonzero *{true)*	nonzero *(true)*	I (true)
nonzero *(true)*	zero *(false)*	I (true)
zero *(false)*	nonzero *{true)*	I (true)
zero *{false)*	zero *(false)*	0 (false)

given that

```
int a, b, c;
a = b = c = 1 0 ;
```

the logical expression

```
a | | (b - c)
```

has the value 1 *(true)*, since a is nonzero *(true)*, and the logical expression

```
(a - b) | |c
```

also has the value 1 *(true)*, since c is nonzero *(true)*, but the logical expression

```
(a - c) | | (b - c)
```

has the value 0 *(false)*, since both (a-c) and (b-c) *are zero (false).*

Logical NOT Operator

The *logical NOT* operator inverts the logical value of an expression and has the following operator formation:

! *exp*

An expression of this form is evaluated by first evaluating the operand. If its value is zero *(false)*, the value of the logical expression is 1 *(true)*; if it is nonzero *(true)*, the value of the logical expression is 0 *(false)*.

Logical NOT operator

exp	! *exp*
nonzero *{true)*	0 *(false)*
zero *{false)*	1 *(true)*

given that

```
int a, b;
a = b = 10;
```

the logical expression

```
! a
```

has the value 0 *(false)*, since a is nonzero *(true)*, and the logical expression

```
! (a - b)
```

has the value 1 *(true)*, since (a - b) is zero *(false)*.

Precedence and associativity of the logical operators

Operators	Type	Associativity
+ - ++ -- !	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right
< <= > >=	Binary	Left to right
== ! =	Binary	Left to right
&&	Binary	Left to right
| |	Binary	Left to right
= *= /= %= += -=	Binary	Left to right

The expression

```
a || b && c
```

is interpreted as

```
a || (b && c)
```

since && has higher precedence than ||.

The expressions

```
! a && b
! a|| b
```

are interpreted as

```
(! a) && b
(! a) || b
```

The expressions

```
a & & to & & c
a | | b | | c
```

are interpreted as

```
(a & & b) & & c
(a | | b) | | c
```

The expressions

```
a < b && c % d
a - b | | c == d
```

are interpreted as

```
(a < b) && (c % d)
(a - b) | | (c == d)
```

the expressions

```
! a > = b & & c / d
a * b | | ! c != d
```

are interpreted as

```
( (! a) >= b ) && (c / d)
(a * b) II ( (! c) ! = d )
```

```
int i = 3, j = 2, k = 1;
```

Expression	Equivalent Expression	Value
! ! k		
! i.== ! j		
k != !k * k		
i > j && j >k		
i != j && j ! = k		
i - j - k | | k == i / j		
i < j | | k < i && j < k		

```
int i = 3, j = 2, k = 1;
```

Expression	Equivalent Expression	Value
! ! k	! (! k)	! (! 1) = ! 0 = 1
! i == ! j	(!i) == (!j)	(!3) == (!2) = 0 == 0 = 1
k != !k * k	k ! = ((!k) * k)	1 != (0 * 1) = 1
i > j && j > k	(i > j) & & (j > k)	1 & & 1 = 1
i != j && j ! = k	(i != j) && (j != k)	1 && 1 = 1
i - j - k | | k == i / j	(i-j-k) || (k == (i/j))	0 || (1 == (3/2)) = 1
i < j | | k < i && j < k	(i<j) || ((k<i) & & (j<k))	0 || (1 && 0) = 0

Evaluation of Logical Expressions

For most operators in C, the order of evaluation is determined by their precedence.

However, logical AND and logical OR operators are always evaluated conditionally from left to right, and this evaluation rule can make a difference when the expression contains a side effect.

Consider, for example, the expression

```
--a || --b && --c
```

and let a = b = c = 10. Following the precedence rule and binding the operands to the operators, we get

((-- a) || ((-- b) && (-- c)))

Hypothetically, if the evaluation order were to follow precedence, && would be evaluated first and we would get

((--a) || (9 && (--c))) and b = 9
((--a) || (9 && 9)) and c = 9
((-- a) || *true*)
(*whatever* | I *true*)
true, or 1

Actually,

the evaluation proceeds from left to right, although && has higher precedence than | |. Evaluating from left to right, we get

(9 | |((--b) && (-- c))) and a = 9
(*true* | | ((--b) && (--c)))
(*true* | | *whatever*)
true, or 1

That is, the expression is again *true*, but now a = 9, b = 10, and c = 10.