Cairo University
Faculty of Computers and Information
Computer Science Department

Programming-1 CS112
2017/2018

Conditionals

Dr. Amin Allam

1 Boolean expressions

A *bool* variable can store one of two values: *true* or *false*. Any integer variable can be treated as boolean variable where the value 0 is equivalent to *false*, and any other value is equivalent to *true*. An *boolean expression* consists of a *relational* or *logical* operator and its operands. It returns the value 1 if it evaluates to true, and 0 if it evaluates to false.

Expression	Returns true when:
`a < b`	a is less than b
`a > b`	a is greater than b
`a == b`	a equals b
`a != b`	a does not equal b
`a <= b`	a is less than or equal b
`a >= b`	a is greater than or equal b
`a && b`	(a is true) and (b is true)
`a || b`	(a is true) or (b is true) or (both are true)
`!a`	a is false

```
int a=17, b=5, c=8, x=6; // Defines and initializes 4 integer variables
bool p, q=false;         // Defines 2 boolean variables and initializes one of them
x = (a > b);       // Stores 1 in x
p = (a >= b);      // Stores true in p
q = a > b;         // Stores true in q
q = true;          // Stores true in q
p = !c;            // Stores false in p
p = -5;            // Stores true in p
q = 0;             // Stores false in q
q = false;         // Stores false in q
x = a <= b;        // Stores 0 in x since a is not less than or equal b
p = a < b;         // Stores false in p
p = (a == b);      // Stores false in p since a does not equal b
                   //  The operator == returns true only if the operands are equal
                   //  The operator = assigns the right value to the left variable
p = a != b;        // Stores true in p since a does not equal b
p = (a>b)&&(b<c);  // Stores true in p since both a>b is true and b<c is true
p = (a!=b)||(b>c); // Stores true in p since a!=b is true
p = (a<b)||(b>c);  // Stores false in p since a<b is false and b>c is false
```

The following table summarizes properties of the previously studied operators (ordered in groups from highest precedence to lowest precedence):

Operator	Description	Associativity	Type	Example
`++ --`	Postfix increment and decrement	left to right	Unary	`a++`
`+ -`	Unary plus and minus	right to left	Unary	`-a`
`++ --`	Prefix increment and decrement	right to left	Unary	`++a`
`!`	Logical NOT	right to left	Unary	`!a`
`(type)`	C-style cast	right to left	Unary	`(double)a`
`* / %`	Multiplication, division, and remainder	left to right	Binary	`a*b`
`+ -`	Addition and subtraction	left to right	Binary	`a+b`
`< <=`	Relational $<$ and $\leq$	left to right	Binary	`a<b`
`> >=`	Relational $>$ and $\geq$	left to right	Binary	`a>b`
`== !=`	Relational $=$ and $\neq$	left to right	Binary	`a==b`
`&&`	Logical AND	left to right	Binary	`a&&b`
`||`	Logical OR	left to right	Binary	`a||b`
`=`	Assignment	right to left	Binary	`a=b`
`*= /= %=`	Compound assignment	right to left	Binary	`a*=b`
`+= -=`	Compound assignment	right to left	Binary	`a+=b`

Assume the expression of the second line of the following code:

```
int a=17, b=5, c=4, d=8; bool x;
x = a <= b || c < d && b >= c;
```

The expression may be evaluated internally by the compiler as follows:

```
int a=17, b=5, c=4, d=8; bool x;
x = a <= b || c < d && b >= c;
x = false  || c < d && b >= c;
x = false  || true  && b >= c;
x = false  || true  && true;
x = false  || true;
x = true;
```

An important property called *short circuit* is explained by the following examples:

```
int a=17, b=5, c=4, d=8; bool x;
x = a > b  || c > d && b <= c;
x = true   || c > d && b <= c;
x = true;        // No need to evaluate the remainder of the expression
```

```
int a=17, b=5, c=4, d=8; bool x;
x = a < b  && c < d && b >= c;
x = false  && c < d && b >= c;
x = false;       // No need to evaluate the remainder of the expression
```

2 The if statement

An *if* statement in the form *if(expression) statement;* will execute `statement` only if `expression` is `true` (or any value other than 0). If `expression` is `false` (or 0), `statement` will not be executed. `statement` can be replaced by a block of statements: {`statement1; statement2; ...statementn;`} all these statements will be executed only if `expression` is `true`. The following example illustrates a simple program that computes the maximum of three integers.

```
#include <iostream>
using namespace std;
int main()
{
   int a, b, c;
   cout << "Enter three integers: ";
   cin >> a >> b >> c;

   int m = a;          // Initially set the current maximum value to a
   if(b > m) m = b;    // If b is larger than the current maximum, update it
   if(c > m) m = c;    // If c is larger than the current maximum, update it

   cout << "The maximum = " << m << endl;
   return 0;
}
```

There are several forms of *if statement* as follows:

- *if(expr) stmt;* Executes `stmt` only if `expr` is true.

- *if(expr) stmt1; else stmt2;*
This is equivalent to: *if(expr) stmt1; if(!expr) stmt2;*
If `expr` is true, `stmt1` is executed. Otherwise, `stmt2` is executed if `expr` is false.

- *if(expr1) stmt1; else if(expr2) stmt2;*
This is equivalent to: *if(expr1) stmt1; else {if(expr2) stmt2;}*
If `expr1` is true, `stmt1` is executed. Otherwise, we check `expr2`.
If `expr1` is false and `expr2` is true, `stmt2` is executed.

- *if(expr1) stmt1; else if(expr2) stmt2; else stmt3;*
This is equivalent to: *if(expr1) stmt1; else {if(expr2) stmt2; else stmt3}*
If `expr1` is true, `stmt1` is executed. Otherwise, we check `expr2`.
If `expr1` is false and `expr2` is true, `stmt2` is executed.
If `expr1` is false and `expr2` is false, `stmt3` is executed.

In all these forms, any `stmt` can be replaced by a block of statements: {`stmt1; stmt2; ...stmtn;`}. Note that the if statement is just a statement: In the last form above, the else statement of the first if is another if statement. Note that in the absence of curly brackets {}, the `else` statement is associated to the last preceding `if` condition.

3 The ternary conditional operator ?:

- *expr1?expr2:expr3;*

If `expr1` is true, the operator `?:` executes and returns the value of `expr2`. Otherwise, the operator executes and returns the value of `expr3`. For example, the following two codes are equivalent:

```
int y, x = 10;
y = x > 9 ? 100 : 200;
```

```
int y, x = 10;
if(x > 9) y = 100; else y = 200;
```

4 The switch statement

```
switch(expr)
{
case constant1: statements1 break;
case constant2: statements2 break;
case constant3: statements3 break;
default: statements4
}
```

This is equivalent to:

```
if(expr==constant1) {statements1}
else if(expr==constant2) {statements2}
else if(expr==constant3) {statements3}
else {statements4}
```

The curly brackets {} around statements are essential for the if statement, while they are not required for the switch statement. The `break;` statement is required in order to end the execution of the switch statement, otherwise, execution continues until the first `break;` encountered, or the end of the switch statement indicated by the curly bracket }. The `default:` case can be removed. The switch statement works when the `expr` evaluates to character or integer value.

```
char a='z'; int b=6, c=7;
if(a=='y')
cout<<b<<endl;                             // Not executed
cout<<c<<endl;                             // Prints 7
if(b==6) {cout<<a<<" "; cout<<c<<endl;}    // Prints z 7
switch(a)
{
   case 'a': cout<<a; break;               // Not executed
   case 'z': cout<<b<<endl;                // Prints 6
   case 'y': cout<<c<<endl;                // Prints 7
}
```