Unit 3 Lesson 7

Building Blocks of Decision Making

With Additions & Modifications by

Mr. Dave Clausen

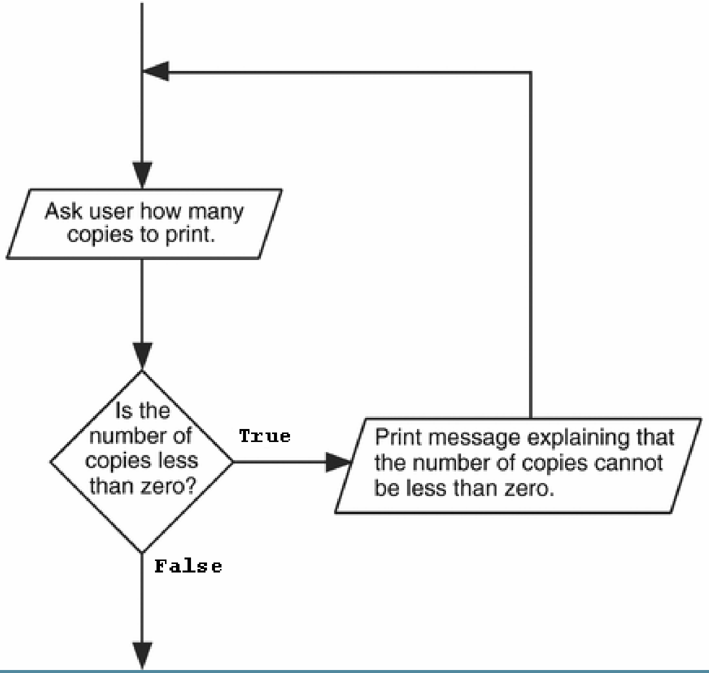
Decision Making in Programs

- Although your brain's method of decision making is more complex than what a computer is capable of, **decision making** in computers **is based on comparing data.**
- In this section you will learn to use the basic tools of computer decision making.
- Almost every program that is useful or user-friendly involves decision making.
- Although some algorithms progress sequentially from the first to last instruction, most algorithms branch out into more than one path.

Decision Making 2

- At the point at which the branching takes place, a decision must be made as to which path to take.
- The flow chart in Figure 7-1 is part of an algorithm in which the program is preparing to output a document to the printer.

FIGURE 7-1
The decision-making part of this flowchart prevents the program from proceeding with invalid data.

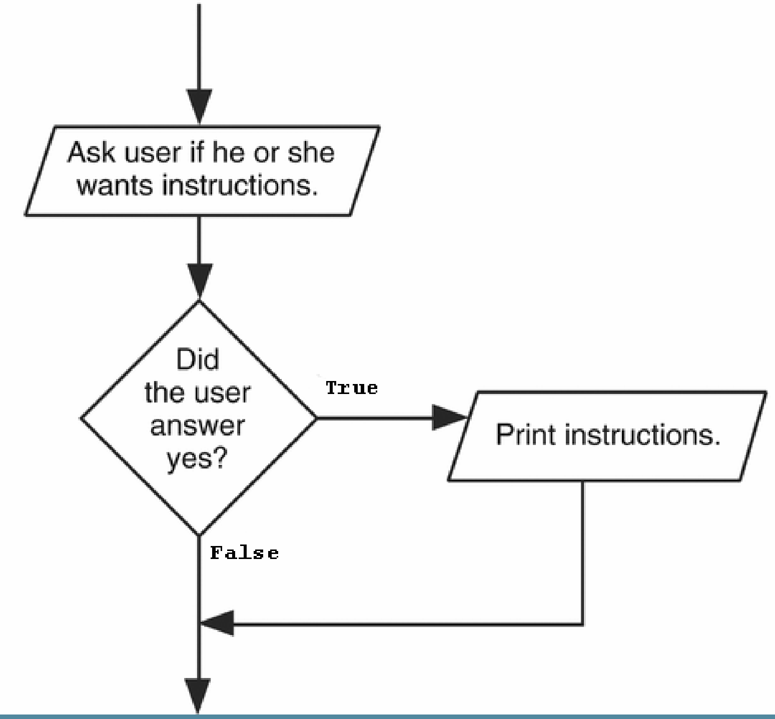

Ask user how many copies to print.
Is the number of copies less than zero?
True
Print message explaining that the number of copies cannot be less than zero.
False

Decision Making 3

- ❖ Decisions may also have to be based on the wishes of the user.
- ❖ The flowchart in Figure 7-2 shows how the response to a question changes the path the program takes.

FIGURE 7-2

The path a program takes may be dictated by the user.



Control Structures

- Corrado Bohm & Guiseppe Jacopini
  - 1964 Structure Theorem

    proved that any program logic, regardless of the complexity, can be expressed using the control structures of **sequencing**, **selection**, and **repetition (iteration)**.

Control Structures 2

- A. Sequence
  - – Instructions executed in order 1st, 2nd, 3rd, etc.
- B. Selection
  - – (Branching, Conditionals)
  - – If, and If else statements
  - – Switch or Case statements

Control Structures 3

- C. Repetition or Iteration
  - – Indefinite Loops
    - while
      - – (condition checked at beginning of the loop)
    - do...while
      - – (condition checked at the end of the loop)
  - – Definite Loops
    - for loop
  - – Recursion

Avoid using GOTO statements

- Edger W. Dijkstra 1968
  - “The GOTO statement should be abolished from all higher level programming languages…”
  - “…The GOTO statement is just too primitive; it is too much of an invitation to make a mess of one’s program.”
- Mr. Clausen “If you use a GOTO statement in your program, you will get a “0” zero on your program.”

Representing True & False in C++

- The way computers make decisions is very primitive.
- Even though computers make decisions similar to the way the human brain does, computers do not have intuition or “gut” feelings.
- Decision making in a computer is based on performing simple comparisons.
- The microprocessor compares two values and “decides” whether the are equivalent.
- Clever programming and the fact that computers can do millions of comparisons per second sometimes make computers appear to be “smart”.

True & False in C++ 2

- When the computer makes a comparison, the comparison results in a value of either 0 or 1.
- If the resulting value is 0, it means the comparison proved **false**.
- If the result is 1, the comparison proved **true**.
- In our source code, we will use the Boolean values of **true** or **false** instead of 0 or 1.
- Good programming style dictates that you never use 0 or 1 in your source code to indicate true or false.

booltest.cpp booltest.txt

Relational Operators

- To make comparisons, C++ provides a set of relational operators, shown in Table 7-1.

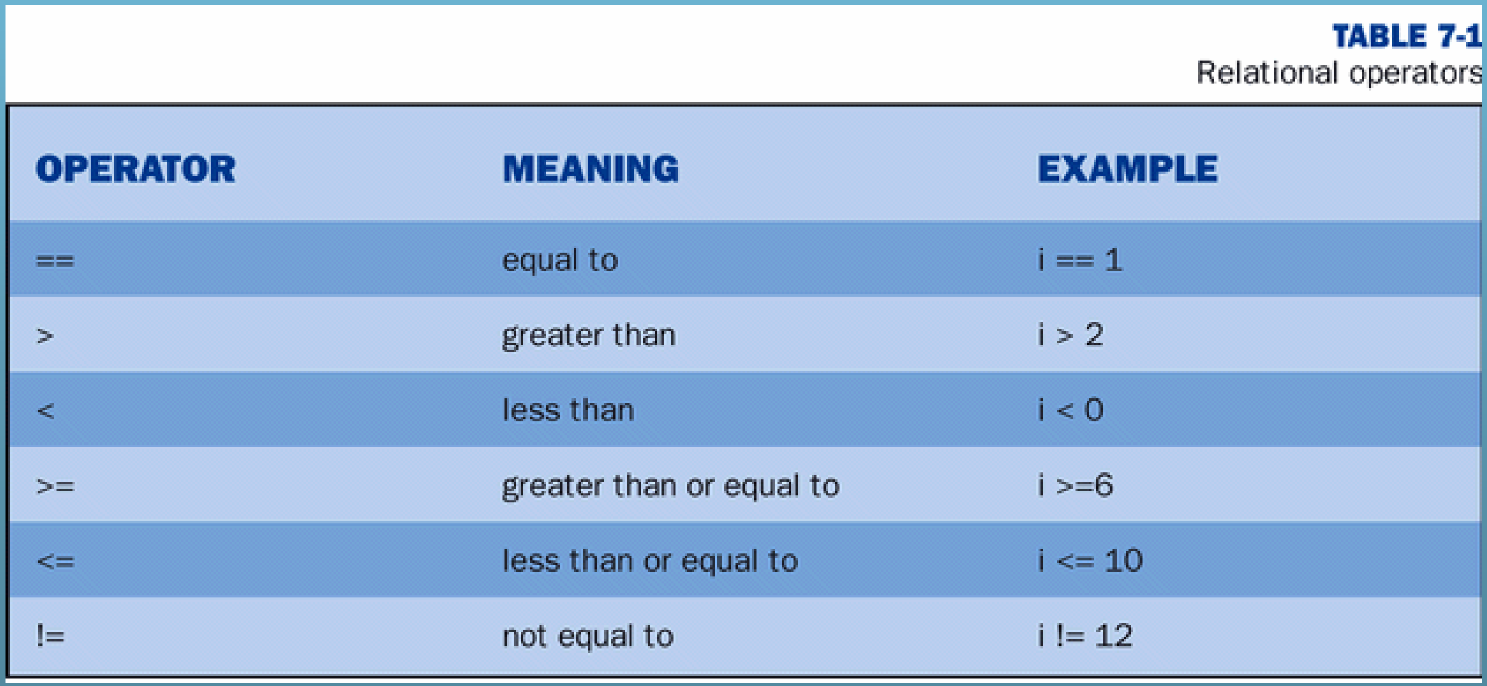


TABLE 7-1
Relational operators

OPERATOR	MEANING	EXAMPLE
==	equal to	i == 1
>	greater than	i > 2
<	less than	i < 0
>=	greater than or equal to	i >=6
<=	less than or equal to	i <= 10
!=	not equal to	i != 12

Relational Operators 2

- They are **similar** to the symbols you have used in math when working with equations and inequalities.
- However, there is one relational operator whose meaning is easily confused if you aren’t careful.
- You will have to remember what works in Math and what works in C++.

Relational Operators 3

Arithmetic Operation	Meaning	Relational Operator
=	Is equal to	= =
<	Is less than	<
>	Is greater than	>
≤	less than or equal to	<=
≥	greater than or equal to	>=
≠	Is not equal to	!=

Relational Operators, and Boolean Expressions

- ❖Relational Operators
  - – Operations used on **same data types** for comparison
    - • equality, inequality, less than, greater than
- ❖Simple Boolean Expression
  - – Two values being compared with a single relational operator
  - – Has a value of true or false

Simple Boolean Expressions

7 = = 7	true
-3.0 = = 0.0	false
4.2 > 3.7	true
-18 < -15	true
13 < 0.013	false
-17.32 != -17.32	false
a = = a	true
a = 7	true

Confusing = and = =

- Don’t confuse the meaning of = in Math with its meaning in C++.
- The symbol = means assigning a value to an identifier, and not that two objects or expressions are “equal” in C++.
- The symbol = = means equality in a comparison in C++.
- Side effects are caused if we confuse the two operators.

P209Ex1.cpp

P209Ex1.txt

Order of Operations

1. (   )
2. *, /, %
3. +, -
4. = =, <, >, <=, >=, !=

Code List 7-1

```
// relate.cpp                              relate.txt

#include <iostream.h>

int main ( )
{
      int i = 2;
      int j = 3;
      bool true_false;

      cout << (i = = 2) << end1; // displays a 1 (true)
      cout << (i = = 1) << end1; // displays a 0 (false)
      cout << (j > i) << end1;
      cout << (j < i) << end1;     // Can you predict
      cout << (j <= 3) << end1; // the output of
      cout << (j >= i) << end1;  // these statements?
      cout << (j != i) << end1;

      true_false = (j < 4); // The result can be stored to a Boolean variable
      cout << true_false << end1;
      return 0;
}
```

Logical Operators

- Sometimes it takes more than two comparisons to obtain the desired results.
  - For example, if you want to test to see whether an integer is in the range 1 to 10, you must do two comparisons.
  - In order for the integer to fall within the range, it must be greater than 0 **AND** less than 11.
- C++ provides three logical operators.

Logical Operators 2

- ❖Table 7-2 shows the three logical operators and their meaning.

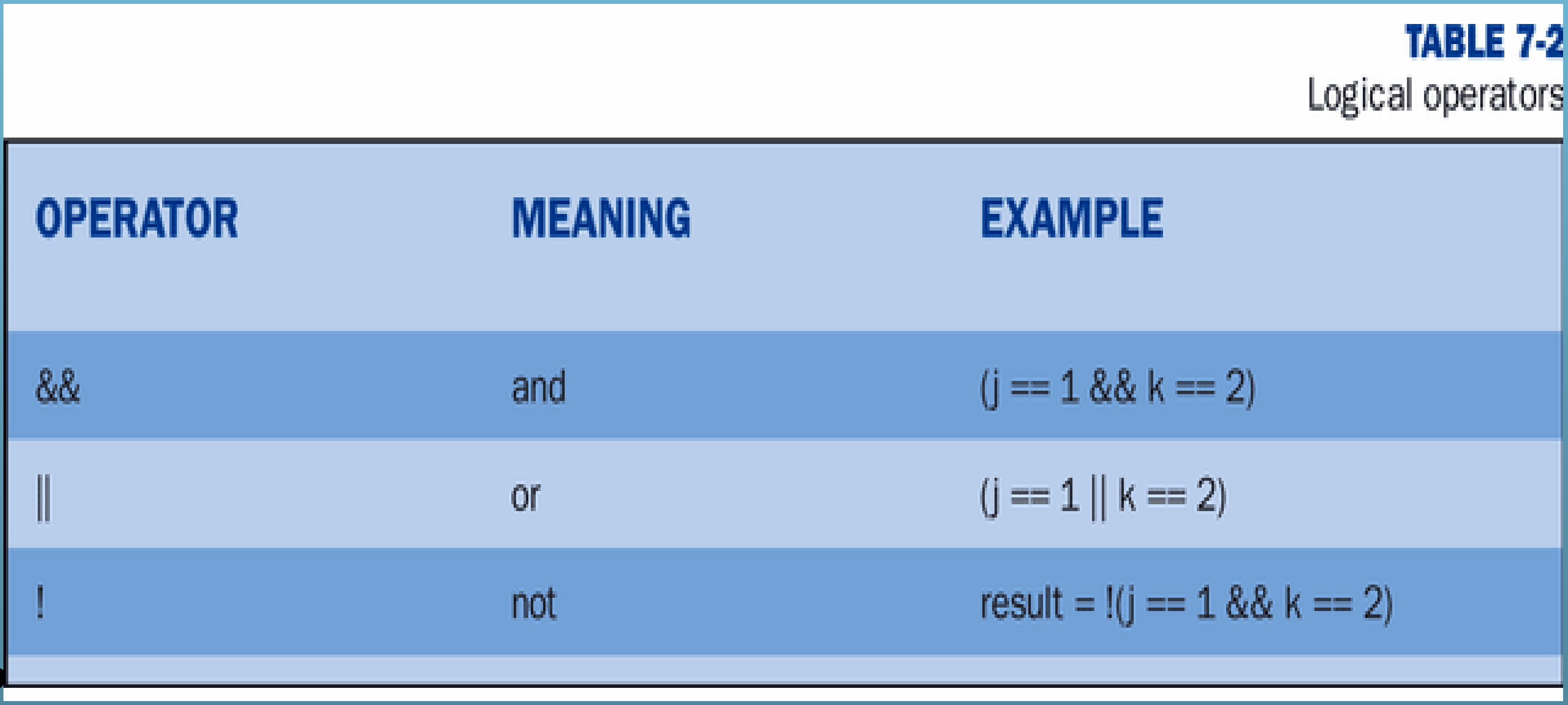


TABLE 7-2
Logical operators

OPERATOR	MEANING	EXAMPLE
&&	and	(j == 1 && k == 2)
||	or	(j == 1 || k == 2)
!	not	result = !(j == 1 && k == 2)

Logical Operators 3

- Consider the following C++ statement.

```
in_range = (i > 0 && i < 11);
```

- The variable in_range is assigned the value 1(true) if the value of i falls into the defined range, and 0 (false) if the value of i does not fall into the defined range.

Logical Operators 4

- The **NOT** operator (**!**) takes the opposite of the stated condition and turns true to false and false to true.
  - Black_and_White = !InColor;
- The **AND** operator (**&&**) requires that both conditions are true for the entire expression to be true.
- In the **OR** operator (||) both expressions would have to be false for the entire expression to be false.

Compound Boolean Expressions

- Logical Operators
  - And && (two ampersands) Conjunction
  - Or || (two pipe symbols) Disjunction
  - Not ! (one exclamation point) Negation
- Place parentheses around each simple expression and another set parentheses around the entire expression for my preferred programming style.
  - i.e. ((grade >= 80) && (grade < 90))
  - While the compiler does not require this style, it will work for all programming languages.

Code List 7-2

```
//logical.cpp                              logical.txt
#include  <iostream.h>
int main ()
{
      int  i =  2;
      int  j  =  3;
      bool true_false;
      true_false  =  ( i  <  3  &&  j  >  3 ) ;
      cout  <<  “The result of  ( i  <  3  &&  j  >  3 )  is “<<  true_false  <<  ‘\n’;
      true_false  =  ( i  <  3  &&  j  >=  3 );
      cout  <<  “The result of  ( i  <  3  &&  j  >=  3 )  is “<<  true_false  <<  ‘\n’;
      cout  <<  “The result of  ( i  ==  1 || i  ==  2 )  is “ <<  ( i  ==  1  | |  i  ==  2 )  <<  ‘\n’;
      true_false  =  ( j  <  4 ) ;
      cout  <<  “the result of  ( j  <  4 )  is “ <<  true_false  <<  ‘\n’;
      return  0;
}
```

Truth Tables

- Figure 7-3 shows three diagrams called truth tables. They will help you understand the result of comparisons with the logical operators and, or, and not.

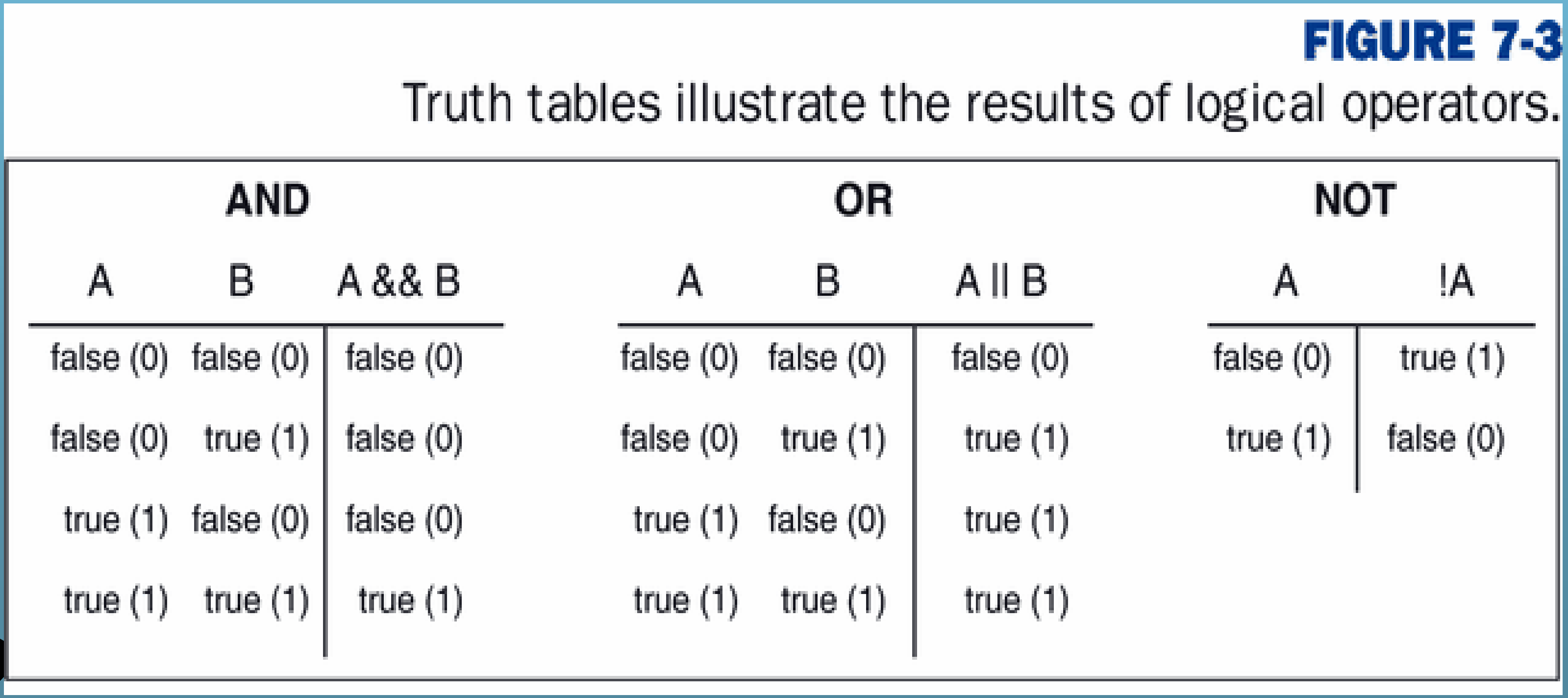


FIGURE 7-3

Truth tables illustrate the results of logical operators.

AND

A	B	A && B
false (0)	false (0)	false (0)
false (0)	true (1)	false (0)
true (1)	false (0)	false (0)
true (1)	true (1)	true (1)

OR

A	B	A II B
false (0)	false (0)	false (0)
false (0)	true (1)	true (1)
true (1)	false (0)	true (1)
true (1)	true (1)	true (1)

NOT

A	!A
false (0)	true (1)
true (1)	false (0)

Combining More Than Two Comparisons

- You can use logical operators to combine more than two comparisons
- Consider the following statement, which decides whether it is okay for a person to ride a roller coaster

```
ok_to_ride  =  ( height _in_inches  >  45  &&
                    !back_trouble &&  !heart_trouble );
```

Order of Logical Operations

- You can mix logical operators in statements as long as you understand the order in which the logical operators will be applied.
- The not operator (!) is applied first, then the and operator (&&), and finally the or operator (||)
- Order Of Priority in Boolean Expressions
  - 1. ! (NOT)
  - 2. && (AND)
  - 3. || (OR)
- Compare the following statements:

```
dog_acceptable = (white || black && friendly); //logic error

dog_acceptable = ((white || black) && friendly); //correct
```

Order of Operations including Relational & Logical Operators

- **1. ( )**
- **2. !**
- **3. *, /, %**
- **4. +, -**
- **5.** <, <=, >, >=, = =, **!=**
- **6. &&**
- **7. ||**

Complements

Operation	Complement (equivalent)
! <	>=
! <=	>
! >	<=
! >=	<

Code List 7-3

```
// logical2.cpp                    logical2.txt
#include <iostream.h>
int main()
{
 bool white, black, friendly, acceptable;
 white = true;       // dog is white
 black = false;      // dog is not black
 friendly = false;  // dog is not friendly
 // The following statement produces incorrect results due to the
 // order of operations.
 acceptable = (white || black && friendly);
 cout << acceptable << endl;
 // The parentheses in the following statement override the
 // order of operations and the statement produces the correct result.
 acceptable = ((white || black) && friendly);
 cout << acceptable << endl;
 return 0;
}
```

Short-Circuit Evaluation

- C++ has a feature called ***short-circuit evaluation*** that allows the compiler to evaluate only a part of your Boolean expression given certain conditions.
- For example, in an expression:

```
in_range = ( i > 0 && i < 11 );
```

- The program first checks to see whether i is greater than 0.
- If it is not, there is no need to check any further because regardless of whether i is less than 11, in_range will be false.
- So the program sets in_range to false and goes to the next statement without evaluating the right side of the && expression.

Short-Circuit Evaluation 2

- Short circuiting also occurs with the **or** (||) operator.
- If the left side of an “**or**” expression is true, the entire expression has a value of true.
- Therefore, there is no need to evaluate the right side of the expression.
- For example:

```
true_false = (i < 5 || i >10);
```