Introduction to Programming with C++

Making Decisions

Objective

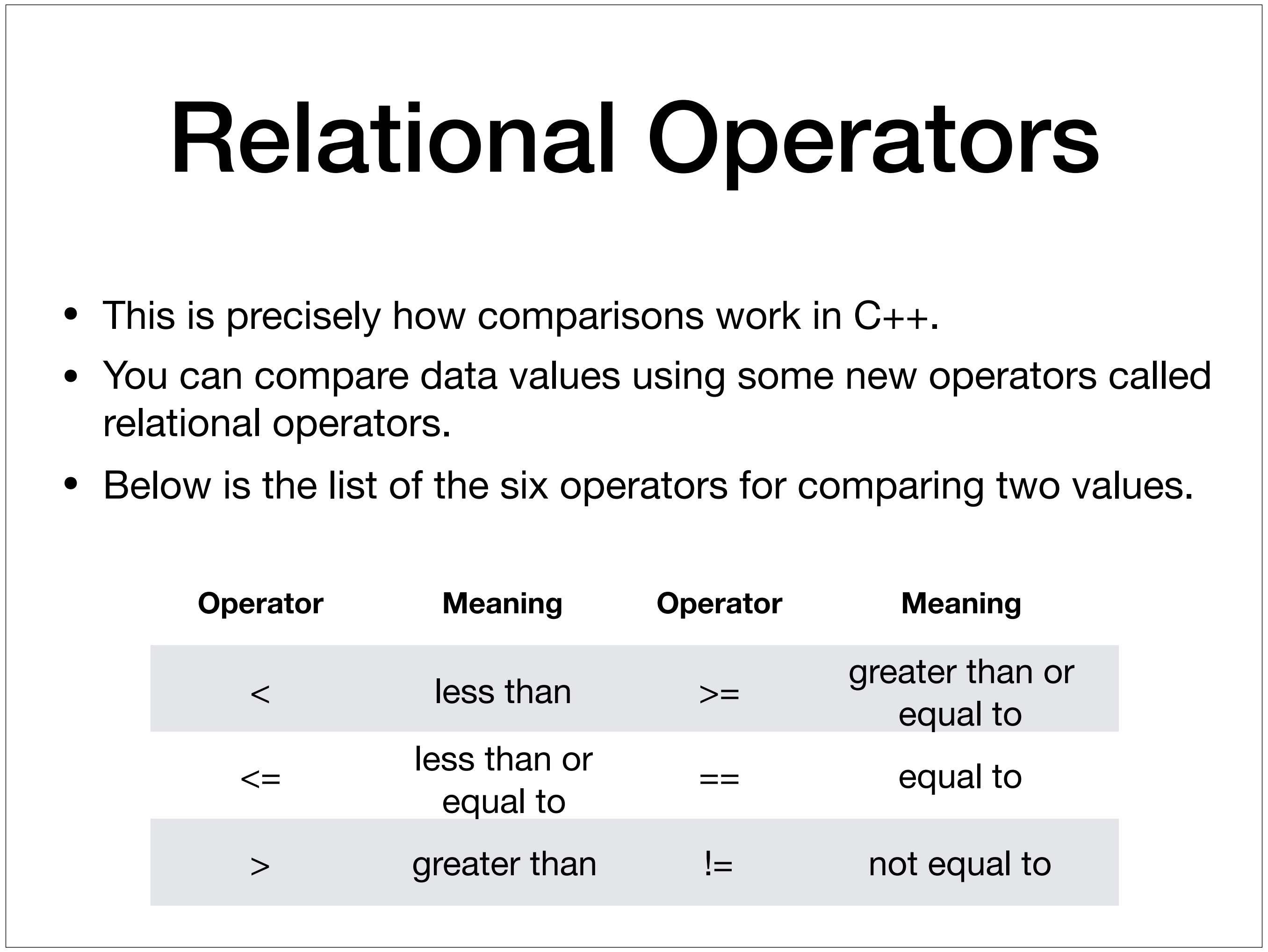
- In this chapter, we’ll explore how to make choices and decisions.
- By the end of this chapter, you will have learned:
  - How to compare data values
  - How to alter the sequence of program execution based on the result of a comparison
  - What logical operators and expressions are, and how you apply them
  - How to deal with multiple-choice situations
  - Data types in C++

Decision Making

- Decision-making is fundamental to any kind of computer programming.
- It’s one of the things that differentiates a computer from a calculator.
- It means altering the sequence of execution depending on the result of a comparison
- This will allow us to validate program input and write programs that can adapt their actions depending on the input data.
- Our programs will be able to handle problems where logic is fundamental to the solution.

Comparing Data Values

- To make decisions, you need a mechanism for comparing things
- There are several kinds of comparisons.
- For instance, “If the traffic signal is red, stop the car,” involves a comparison for equality.
- You compare the color of the signal with a reference color, red, and if they are equal, you stop the car.
- On the other hand “If the speed of the car exceeds the limit, slow down,” involves a different relationship.
- Here you check whether the speed of the car is greater than the current speed limit.
- Both of these comparisons are similar in that they result in one of two values:
    - They are either true or false.



Relational Operators

- This is precisely how comparisons work in C++.
- You can compare data values using some new operators called relational operators.
- Below is the list of the six operators for comparing two values.

Operator	Meaning	Operator	Meaning
<	less than	>=	greater than or equal to
<=	less than or equal to	==	equal to
>	greater than	!=	not equal to

Caution T he equal to operator, ==, has two successive equal signs. It’s a very common mistake to use one equal sign instead of two to compare for equality. This will not necessarily result in a warning message from the compiler because the expression may be valid but just not what you intended, so you need to take particular care to avoid this error.

Boolean Values

- Each of these operators compares two values and results in a value of type bool;
- there are only two possible bool values, true and false.
- true and false are keywords and are literals of type bool/ Boolean literals
- If you cast true to an integer type, the result will be 1; casting false to an integer results in 0.
- You can also convert numerical values to type bool.
- Zero converts to false, and any nonzero value converts to true.

Defining Integer Variables

- When you have a numerical value where a bool value is expected, the compiler will insert an implicit conversion to convert the numerical value to type bool.
- This is very useful in decision-making code.
- You create variables of type bool just like other fundamental types. Here's an example:

```
bool isValid {true};
```

- This defines the variable isValid as type bool with an initial value of true.

Applying the Comparison Operators

- You can see how comparisons work by looking at a few examples.
- Suppose you have integer variables i and j, with values 10 and –5 respectively.
- Consider the following expressions:

```
i > j
i != j
j > -8
i <= j + 15
```

- All of these expressions evaluate to `true`.
- Note that in the last expression, the addition, j + 15, executes first because + has a higher precedence than <=.

Applying the Comparison Operators

- You could store the result of any of these expressions in a variable of type bool. For example:

```
isValid = i > j;
```

- If i is greater than j, true is stored in isValid, otherwise false is stored.
- You can compare values stored in variables of character types, too.
- Assume that you define the following variables:

```
char first {'A'};
char last {'Z'};
```

Applying the Comparison Operators

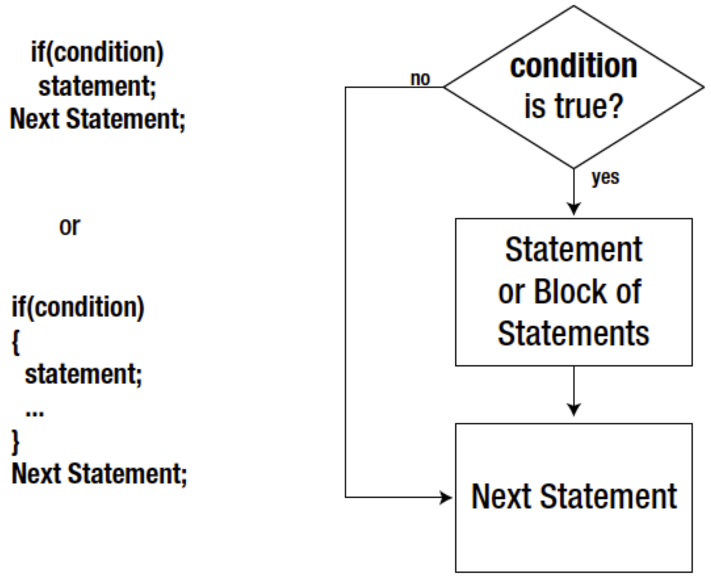
- You can write comparisons using these variables:

```
first < last
'E' <= first
first != last
```

- Here you are comparing code values.
- The first expression checks whether the value of first is less than the value of last.
- This is always true.
- The result of the second expression is false.
- The last expression is true, because `'A'` is definitely not equal to `'Z'`.

The if Statement

- The basic if statement enables you to choose to execute a single statement, or a block of statements when a given condition is true



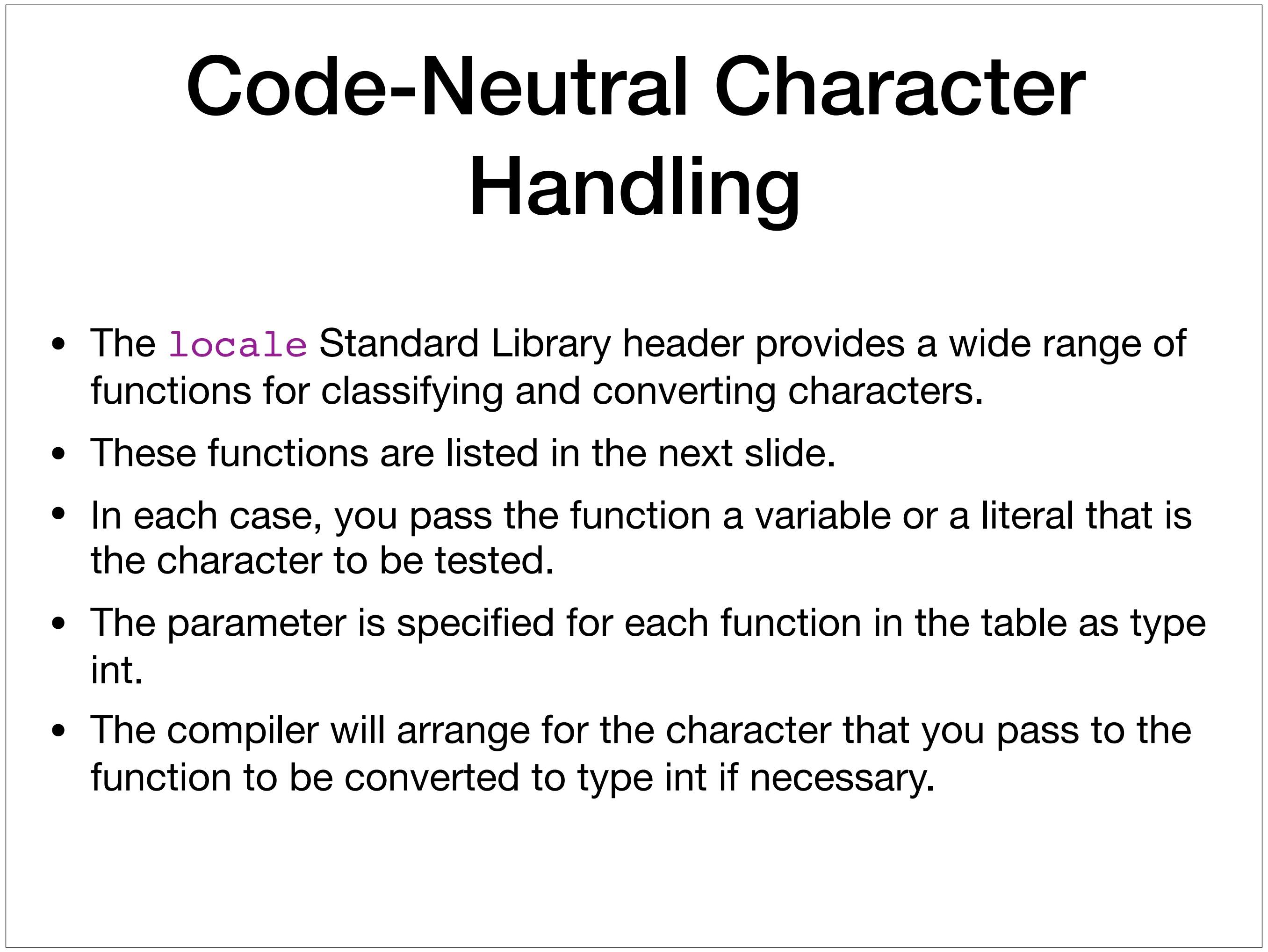
```
if(condition)
  statement;
Next Statement;


      or


if(condition)
{
  statement;
  ...
}
Next Statement;
```

See Ex4_02.cpp

Nested if Statements

- The statement that executes when the condition in an if statement is true can itself be an if statement.
- This arrangement is called a nested if.
- The condition of the inner if is only tested if the condition for the outer if is true.
- An if that is nested inside another can also contain a nested if.
- You can nest ifs to whatever depth you require.

See Ex4_03.cpp



Code-Neutral Character Handling

- The `locale` Standard Library header provides a wide range of functions for classifying and converting characters.
- These functions are listed in the next slide.
- In each case, you pass the function a variable or a literal that is the character to be tested.
- The parameter is specified for each function in the table as type int.
- The compiler will arrange for the character that you pass to the function to be converted to type int if necessary.

- The `cctype` and the `cwctype` headers that are inherited from C are also part of the C++ Standard Library.
- The `cctype` header declares the same classification and conversion functions that we have described for the locale header.
- The `cwctype` header declares an equivalent set of functions with slightly different names that work with characters of type `wchar_t`.

Function	Operation
`isupper(int c)`	Tests whether or not c is an uppercase letter, by default 'A' to 'Z'.
`islower(int c)`	Tests whether or not c is a lowercase letter, by default 'a' to 'z'.
`isalpha(int c)`	Tests whether or not c is an upper- or lowercase letter.
`isdigit(int c)`	Tests whether or not c is a digit, 0 to 9.
`isxdigit(int c)`	Tests whether or not c is a hexadecimal digit, 0 to 9, 'a' to 'f', or 'A' to 'F'.
`isalnum(int c)`	Tests whether or not c is a letter or a digit (i.e., an alphanumeric character).

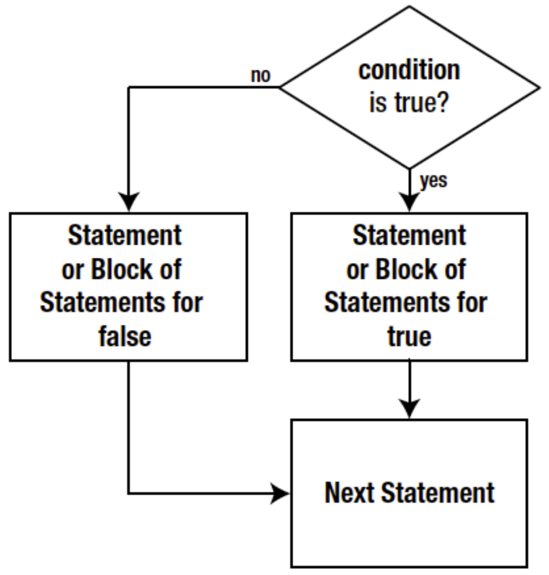
Function	Operation
`isspace(int c)`	Tests whether or not c is whitespace, which can be a space, a newline, a carriage return, a form feed, or a horizontal or vertical tab.
`iscntrl(int c)`	Tests whether or not c is a control character.
`isprint(int c)`	Tests whether or not c is a printable character, which can be an upper- or lowercase letter, a digit, a punctuation character, or a space.
`isgraph(int c)`	Tests whether or not c is a graphic character, which is any printable character other than a space.
`ispunct(int c)`	Tests whether or not c is a punctuation character, which is any printable character that's not a letter or a digit. This will be either a space or one of the following: _ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '.

The if-else Statement

- The if statement that you have been using executes a statement or block if the condition specified is true.
- Program execution then continues with the next statement in sequence.
- Of course, you may want to execute one block of statements when the condition is true, and another set when the condition is false.
- An extension of the if statement called an if-else statement allows this.

The if-else statement logic

```
if( condition )
{
  // Statements when condition is true
}
else
{
  // Statements when condition is false
}
// Next Statement
```


condition is true?
no
yes
Statement or Block of Statements for false
Statement or Block of Statements for true
Next Statement

One of the two blocks in an if-else statement is always executed.

See Ex4_04.cpp

Nested if-else Statements

- You have already seen that you can nest if statements within if statements.
- You can also nest `if-else` statements within `if`s, ifs within `if-else` statements and `if-else` statements within `if-else` statements
- This provides you with plenty of versatility

```
if(coffee == 'y')
if(donuts == 'y')
    std::out << "We have coffee and donuts.”
             << std::endl;
else
  std::cout << "We have coffee, but not donuts."
            << std::endl;
```

Fixed Value Variables

- As you have seen, using ifs where you have two or more related conditions can be cumbersome.
- You have tried your iffy talents on looking for coffee and donuts, but in practice, you may want to check much more complex conditions.
- Defining a test for this could involve the mother of all ifs.
- The logical operators provide a neat and simple solution.
- You can combine a series of comparisons into a single expression using logical operators
- This way you need just one if, almost regardless of the complexity of the set of conditions.

Logical Operators

Operator	Description
&&	Logical AND
||	Logical OR
!	Logical negation (NOT)

See Ex4_06.cpp

The Conditional Operator

- Sometimes called the ternary operator because it involves three operands.
- It parallels the if-else statement
- Instead of selecting one of two statement blocks to execute depending on condition, it selects the value of one of two expressions.
- Thus the conditional operator enables you to choose between two values.
- Suppose you have two variables, a and b, and you want to assign the value of the greater of the two to a third variable, c.
- The following statement will do this

The Conditional Operator

```
c = a > b ? a : b; // Set c to the higher of a and b
```

- The conditional operator has a logical expression as its first operand, in this case a > b.
- If this expression is true, the second operand is selected as the value resulting from the operation.
- If the first operand is false, the third operand is selected as the value.
- Thus, the result of the conditional expression is a if a is greater than b, and b otherwise. This value is stored in c.

The Conditional Operator

- The assignment statement is equivalent to the if statement:

```
if(a > b) {
    c = a;
} else {
    c = b;
}
```

See Ex4_06.cpp

The switch Statement

- You’re often faced with a multiple-choice situation
- Meaning you need to execute a particular set of statements from a number of choices   depending on the value of an integer variable or expression.
- The switch statement enables you to select from multiple choices.
- The choices are identified by a set of fixed integer values and the selection of a particular choice is determined by the value of a given integer expression.
- The choices in a switch statement are called cases.
- A lottery where you win a prize depending on your number coming up is an example of where it might apply.

The switch Statement

- You buy a numbered ticket, and if you’re lucky, you win a prize.
- For instance
  - if your ticket number is 147, you win first prize
  - if it’s 387 you can claim second prize
  - ticket number 29 gets you third prize
  - any other ticket number wins nothing.
- The switch statement to handle this situation would have four cases
  - one for each of the winning numbers, plus a “default” case for all the losing numbers.

Switch Example

```
switch(ticketNumber) {
        case 147:
             std::cout << "You win first prize!”;
             break;
        case 387:
             std::cout << "You win second prize!";
             break;
        case 29:
             std::cout << "You win third prize!";
             break;
        default:
             std::cout << "Sorry, you lose.";
             break;
}
```

Switch Explained

- The possible choices in a switch statement appear in a block, and each choice is identified by a case value.
- A case value appears in a case label, which is of the form:
- case case_value:
- It’s called a case label because it labels the statements or block of statements that it precedes.
- The statements that follow a particular case label execute if the value of the selection expression is the same as that of the case value.
- Each case value must be unique but case values don’t need to be in any particular order, as the example demonstrates.

Switch Explained

- Each case value must be an integer constant expression, which is an expression that the compiler can evaluate
- this implies that it can only involve literals, or const variables.
- Furthermore, any literals must either be of an integer type or be able to be converted to an integer type.
- The default label in the example identifies the default case, which is a catchall that is selected if none of the other cases is selected.
- You don’t have to specify a default case, though.
- If you don’t, and none of the case values is selected, the switch does nothing.
- The break statement that appears after each set of case statements is essential for the logic here.

Switch Explained

- Executing a break statement breaks out of the switch and causes execution to continue with the statement following the closing brace.
- If you omit the break statement for a case, the statements for the following case will execute.
- Notice that we don't need a break after the final case (usually the default case) because execution leaves the switch at this point anyway.
- It's good programming style to include it though because it safeguards against accidentally falling through to another case that you might add to a switch later.

See Ex4_07.cpp and Ex4_08.cpp

Unconditional Branching

- The `goto` statement is a blunt instrument.
- It enables you to branch to a specified program statement unconditionally.
- The statement to be branched to must be identified by a statement label
- A label is an identifier defined according to the same rules as a variable name.
- This is placed before the statement to be referenced and separated from it by a colon. Here's an example of a labeled statement:

```
MyLabel: x = 1;
```

- This statement has the label MyLabel, and an unconditional branch to this statement would be written as follows:

```
goto MyLabel;
```

- Whenever possible, you should avoid using goto statements.
- They encourage convoluted code that can be extremely difficult to follow.

Summary

- In this chapter, you have added the capability for decision-making to your programs.
- You now know how all the decision-making statements in C++ work.
- The essential elements of decision-making that you have learned about in this chapter are:
    - You can compare two values using the comparison operators.
    - This will result in a value of type bool, which can be true or false.
    - You can convert a bool value to an integer type—true will convert to 1 and false will convert to 0.
    - Numerical values can be converted to type bool—a zero value converts to false, and a nonzero value casts to true.

Summary

- When a numerical value appears where a boll value is expected - such as in an if condition - the compiler will insert an implicit conversion of the numerical value to type bool.
- The if statement executes a statement or a block of statements depending on the value of a condition expression.
- If the condition is true, the statement or block executes.
- If the condition is false it doesn’t.
- The if-else statement executes a statement or block of statements when the condition is true, and another statement or block when the condition is false.

Summary

- if and if-else statements can be nested.
- The switch statement provides a way to select one from a fixed set of options, depending on the value of an integer expression.
- The conditional operator selects between two values depending on the value of an expression.
- You can branch unconditionally to a statement with a specified label by using a goto statement.

Exercise

1. Write a program that prompts for two integers to be entered and then uses an if-else statement to output a message that states whether or not the integers are the same.
2. Create a program that prompts for input of an integer between 1 and 100. Use a nested if, first to verify that the integer is within this range, and then, if it is, to determine whether or not the integer is greater than, less than, or equal to 50. The program should output information about what was found.

Exercise

1. Design a program that prompts for input of a letter. Use a library function to determine whether or not the letter is a vowel and whether it is lowercase or not, and output the result. Finally, output the lowercase letter together with its character code as a binary value.
2. Write a program that determines, using only the conditional operator, if an integer that is entered has a value that is 20 or less, is greater than 20 and not greater than 30, is greater than 30 but not exceeding 100, or is greater than 100.
3. Create a program that prompts the user to enter an amount of money between $0 and $10 (decimal places allowed). Determine how many quarters (25c), dimes (10c), nickels (5c), and pennies (1c) are needed to make up that amount. Output this information to the screen and ensure that the output makes grammatical sense (for example, if you need only one dime then the output should be “1 dime” and not “1 dimes”).