Selection Structures

Chapter 4

Objectives

- Compare values of primitive types using Boolean expressions
- Use C branching statements
  - `if-else`
  - `switch-case`

Flow of control

- *Flow of control* is the order in which a program performs actions.
  - Up to this point, the order has been sequential.
- A *branching statement* chooses between two or more possible actions.
- A *loop statement* repeats an action until a stopping condition occurs.

Introduction to Boolean Expressions

- The value of a *Boolean expression* is either `1 (true)` or `0 (false)`.
- Examples

```
time < limit
balance <= 0
```

C Comparison Operators

- C Comparison Operators

Notation		Notation	
=	Equal to	==	balance == 0 answer == 'y'
≠	Not equal to	!=	income != tax answer != 'y'
>	Greater than	>	expenses > income
≥	Greater than or equal to	>=	points >= 60
<	Less than	<	pressure < max
≤	Less than or equal to	<=	expenses <= income

Compound Boolean Expressions

- Boolean expressions can be combined using the "and" `(&&)` operator.
- Example

```
if ((score > 0) && (score <= 100))
   ...
```

- Not allowed

```
if (0 < score <= 100)
   ...
```

Compound Boolean Expressions

- Boolean expressions can be combined using the "or" `(||)` operator.
- Example

```
if ((quantity > 5) || (cost < 10))
...
```

- Syntax

```
(Sub_Expression_1) ||
  (Sub_Expression_2)
```

Compound Boolean Expressions

- The larger expression is true
  - When either of the smaller expressions is true.
  - When both of the smaller expressions are true.
- The C version of "or" is the *inclusive or* which allows either or both to be true.
- The *exclusive or* allows one or the other, but not both to be true.

Negating a Boolean Expression

- A boolean expression can be negated using the "not" (`!`) operator.
- Syntax

```
!(Boolean_Expression)
```

- Example

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

which is the *exclusive or*

The `if-else` Statement: Outline

- Basic `if-else` Statement
- Compound `if-else` statements
- Nested `if-else` Statements
- Multibranch `if-else` Statements
- The `switch` Statament
- (optional) The Conditional Operator
- The `exit` Method

Negating a Boolean Expression

- Avoiding the Negation Operator

! (*A Op B*) Is Equivalent to (*A Op B*)	
<	>=
<=	>
>	<=
>=	<
==	!=
!=	==

C Logical Operators

- AND, OR and NOT

Logical *and*	&&	`(sum > min) && (sum < max)`
Logical *or*	||	`(answer == 'y') || (answer == 'Y')`
Logical *not*	!	`!(number < 0)`

Boolean Operators

- The Effect of the Boolean Operators `&&` (and), `||` (or), and `!` (not) on Boolean values

Value of *A*	Value of *B*	Value of *A* && *B*	Value of *A* || *B*	Value of ! (*A*)
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Using ==

- `==` is appropriate for determining if two integers or characters have the same value.

```
if (a == 3)
```

where `a` is an integer type

- `==` is **not** appropriate for determining if two floating points values are equal. Use `<` and some appropriate tolerance instead.

```
if (abs(b - c) < epsilon)
```

where `b`, `c`, and `epsilon` are floating point types

Using ==

- `==` is not appropriate for determining if two strings or arrays have the same value.
  - `if (s1 == s2),` where `s1` and `s2` refer to strings, determines only if s1 and s2 refer the a common memory location.
  - If `s1` and `s2` refer to strings with identical sequences of characters, but stored in different memory locations, `(s1 == s2)` is false.
  - There are special string comparison functions to compare strings for equality

Lexicographic Order

- Characters are compared by lexicographical order
- Lexicographic order is similar to alphabetical order, but is it based on the order of the characters in the ASCII (and Unicode) character set.
  - All the digits come before all the letters.
  - All the uppercase letters come before all the lower case letters.

Boolean expressions

- A Boolean expression has two outcomes with only two values: `1 (true)` and `0 (false)` .
- Can use a single value to evaluate a decision.

```
if (systemsAreOK)
```

instead of

```
if((temperature <= 100) && (thrust
  >= 12000) && (cabinPressure > 30)
  && …)
```

Boolean Expressions and Variables

- Variables, constants, and expressions all evaluate to either `1 (true)` or `0 (false)` .
- A integer variable can be given the value of a Boolean expression by using an assignment operator.

```
int isPositive = (number > 0);
...
if (isPositive) ...
```

Naming Variables from Boolean Expressions

- Choose names such as `isPositive` or `systemsAreOk`.
- Avoid names such as `numberSign` or `systemStatus`.

Precedence Rules

- Parentheses should be used to indicate the order of operations.
- When parentheses are omitted, the order of operation is determined by *precedence rules.*
- Operations with *higher precedence* are performed before operations with *lower precedence.*
- Operations with *equal precedence* are done left-to-right (except for unary operations which are done right-to-left).

Precedence Rules

- High to Low

Highest Precedence

First: the unary operators +, –, ++, ––, and !

Second: the binary arithmetic operators *, /, %

Third: the binary arithmetic operators +, –

Fourth: the boolean operators <, >, <=, >=

Fifth: the boolean operators ==, !=

Sixth: the boolean operator &

Seventh: the boolean operator |

Eighth: the boolean operator &&

Ninth: the boolean operator ||

Lowest Precedence

Short-circuit Evaluation

- Sometimes only part of a boolean expression needs to be evaluated to determine the value of the entire expression.
  - If the first operand associated with an `||` is `true`, the expression is `true`.
  - If the first operand associated with an `&&` is `false`, the expression is `false`.
- This is called *short-circuit* or *lazy* evaluation.

Short-circuit Evaluation

- Short-circuit evaluation is not only efficient, sometimes it is essential!
- A run-time error can result, for example, from an attempt to divide by zero.

```
if ((number != 0) && (sum/number > 5))
```

Precedence Rules

- In what order are the operations performed?

```
score < min/2 - 10 || score > 90
score < (min/2) - 10 || score > 90
score < ((min/2) - 10) || score > 90
(score < ((min/2) - 10)) || score > 90
(score < ((min/2) - 10)) || (score > 90)
```

The `if-else` Statement

- A branching statement that chooses between two possible actions.
- Syntax

```
if (Boolean_Expression)
  Statement_1
else
  Statement_2
```

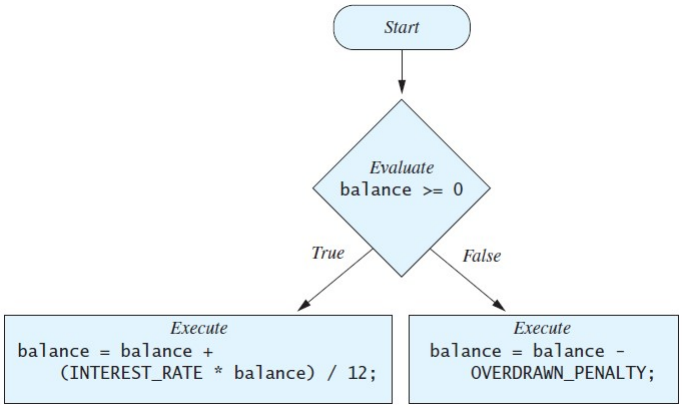
The `if-else` Statement

- Example

```
if (balance >= 0)
   balance = balance + (INTEREST_RATE * balance) / 12;
else
   balance = balance - OVERDRAWN_PENALTY;
```

The `if-else` Statement

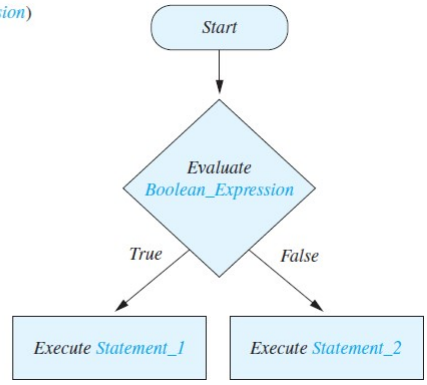
- The Action of the `if-else`



Semantics of the `if-else` Statement

- Two options with if and else

```
if (Boolean_Expression)
    Statement_1
else
    Statement_2
```



Compound Statements

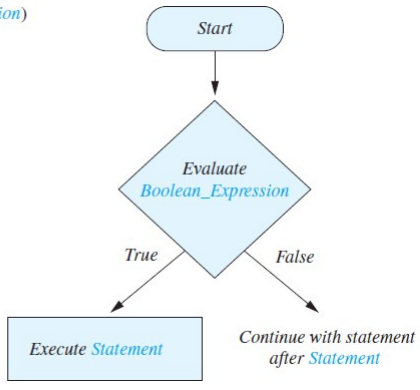
- To include multiple statements in a branch, enclose the statements in braces.

```
if (count < 3)
    {
       total = 0;
       count = 0;
    }
```

Omitting the `else` Part

- The Semantics of an `if` Statement without an `else`

```
if (Boolean_Expression)
    Statement
```



Nested `if-else` Statements

- An `if-else` statement can contain any sort of statement within it.
- In particular, it can contain another `if-else` statement.
  - An `if-else` may be nested within the "if" part.
  - An `if-else` may be nested within the "else" part.
  - An `if-else` may be nested within both parts.

Nested Statements

- Syntax

```
if (Boolean_Expression_1)
    if (Boolean_Expression_2)
       Statement_1)
    else
       Statement_2)
else
   if (Boolean_Expression_3)
       Statement_3)
    else
       Statement_4);
```

Nested Statements

- Each `else` is paired with the nearest unmatched `if`.
- **If used properly**, indentation communicates which `if` goes with which `else`.
- Braces can be used like parentheses to group statements.

Nested Statements

- Subtly different forms

First Form

```
if (a > b)
{
    if (c > d)
        e = f
}
    else
        g = h;
```

Second Form

```
if (a > b)
   if (c > d)
        e = f
    else
        g = h;

// oops
```

Compound Statements

- When a list of statements is enclosed in braces ({}), they form a single *compound statement.*
- Syntax

```
{
    Statement_1;
    Statement_2;
   …
}
```

Compound Statements

- A compound statement can be used wherever a statement can be used.
- Example

```
if (total > 10)
{
   sum = sum + total;
   total = 0;
}
```

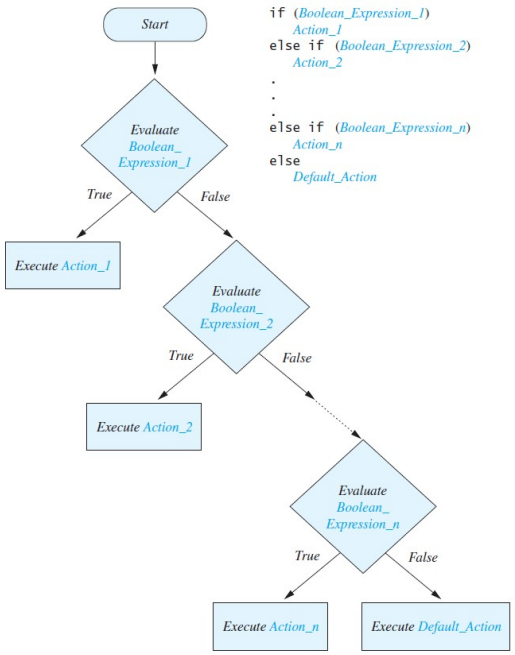
Multibranch `if-else` Statements

- Syntax

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
else if (Boolean_Expression_3)
    Statement_3
else if …
else
    Default_Statement
```

Multibranch `if-else` Statements

- Semantics



Multibranch `if-else` Statements

- Equivalent code

```
if (score >= 90)
    grade = 'A';
else if ((score >= 80) && (score < 90))
    grade = 'B';
else if ((score >= 70) && (score < 80))
    grade = 'C';
else if ((score >= 60) && (score < 70))
    grade = 'D';
else
    grade = 'F';
```

Case Study – Body Mass Index

- Body Mass Index (BMI) is used to estimate the risk of weight-related problems
- BMI = mass / height^2
  - Mass in kilograms, height in meters
- Health assessment if:
  - BMI < 18.5 Underweight
  - 18.5 ≤ BMI < 25 Normal weight
  - 25 ≤ BMI < 30 Overweight
  - 30 ≤ BMI Obese

Case Study – Body Mass Index

- Algorithm
  - Input height in feet & inches, weight in pounds
  - Convert to meters and kilograms
    - 2.2 lb = 1 kg
    - 1 inch = 0.0254 meters
  - Compute BMI
  - Output health risk using if statements
  - See program for code example

Input Validation

- You should check your input to ensure that it is within a valid or reasonable range. For example, consider a program that converts feet to inches. You might write the following:

```
int feet = ?;
int inches = feet * 12;
```

- What if:
  - The user types a negative number for feet?
  - The user enters an unreasonable value like 100? Or a number larger than can be stored in an int? (2,147,483,647)

Input Validation

- Address these problems by ensuring that the entered values are reasonable:

```
int feet;
printf("Enter feet: ");
scanf("%d", &feet);
if ((feet >= 0) && (feet < 10))
{
  int inches = feet * 12;
  ...
}
```

The `switch` Statement

- The **`switch`** statement is a mutlti-way branch that makes a decision based on an *integral* (integer or character) expression.
  - Java 7 allows String expressions
- The **`switch`** statement begins with the keyword **`switch`** followed by an integral expression in parentheses and called the *controlling expression.*

The `switch` Statement

- A list of cases follows, enclosed in braces.
- Each case consists of the keyword `case` followed by
  - A constant called the *case label*
  - A colon
  - A list of statements.
- The list is searched for a case label matching the controlling expression.

The `switch` Statement

- The action associated with a matching case label is executed.
- If no match is found, the case labeled `default` is executed.
  - The `default` case is optional, but recommended, even if it simply prints a message.
- Repeated case labels are not allowed.

The `switch` Statement

- Syntax

```
switch (Controlling_Expression)
{
   case Case_Label:
        Statement(s);
        break;
   case Case_Label:
   …
   default:
   …
}
```

The `switch` Statement

- Number of babies

```
switch(num){
   case 1:
      printf("Congratz one baby!\n");
      break;
   case 2:
      printf("Congratz twins!\n");
      break;
   case 3:
      printf("Wow triplets!");
      break;
   …
   default:
      printf("Huh?");
}
```

The `switch` Statement

- The action for each case typically ends with the word `break`.
- The optional `break` statement prevents the consideration of other cases.
- The controlling expression can be anything that evaluates to an integral type.

Summary

- You have learned about branching statements.
- You have learned about the Boolean expressions.