Programming Fundamentals I

Chapter 4:

Control Structures I (Selection)

Dr. Adriana Badulescu

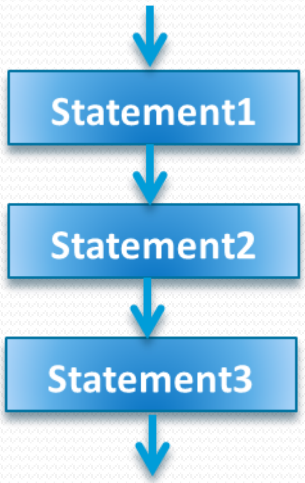
Objectives

- Learn about control structures
- Examine relational and logical operators
- Explore how to form and evaluate logical (Boolean) expressions
- Discover how to use the selection control structures if, if...else, and switch in a program
- Learn how to avoid bugs/errors
- Learn to use the assert function to terminate a program

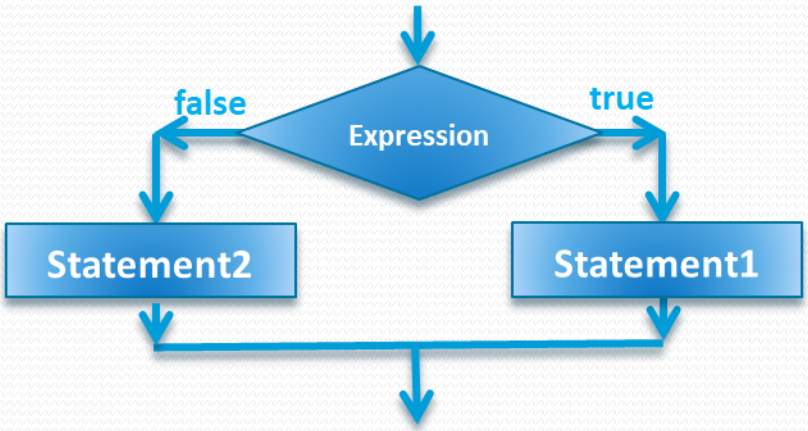
Control Structures

- A computer can proceed:
  - In sequence
  - Selectively (branch): making a choice
  - Repetitively (iteratively): looping
- Some statements are executed only if certain conditions are met
- A condition is met if it evaluates to `true`

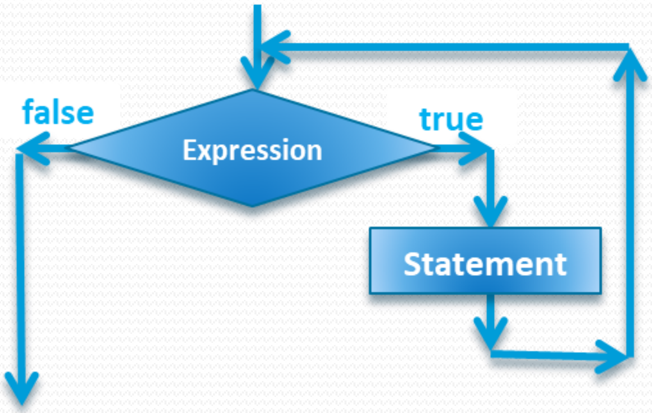
Control Structures



Sequence



Selection



Repetition

Relational Operators

- A **condition** is represented by a logical (Boolean) expression that can be `true` or `false`
- **Relational operators**:
  - Allow comparisons between two operands (binary)
  - Evaluate to `true` or `false`

Operator	Description
==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

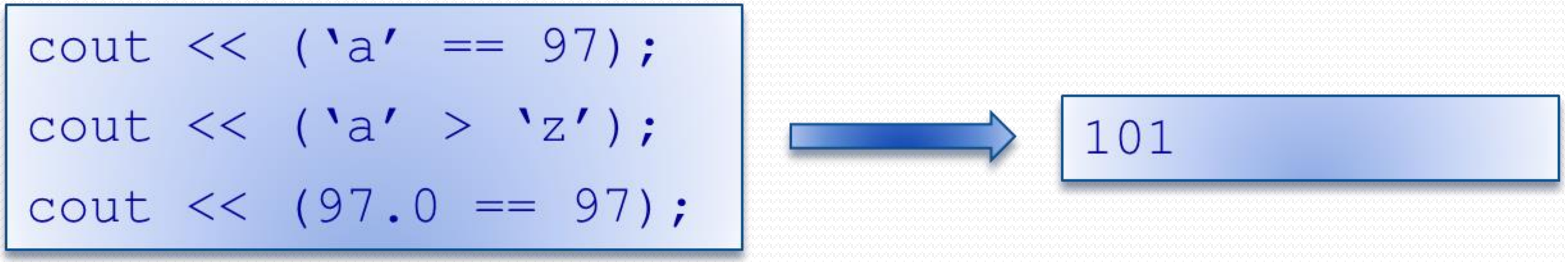
Relational Operators and Simple Data Types

- You can use the relational operators with all three simple data types:

`(8 < 15)`	evaluates to `true`
`(6 != 6)`	evaluates to `false`
`(2.5 > 5.8)`	evaluates to `false`
`(5.9 <= 7.5)`	evaluates to `true`
`(‘a’ == 97)`	evaluates to `true`
`(97.0 == 97)`	evaluates to `true`
`(‘A’ > ‘a’)`	evaluates to `false(65<97)`
`(a = ‘0’)`	evaluates to `true`

Relational Operators and Simple Data Types

- **Logical (Boolean) expressions**
  - Expression with relational operators
  - Returns an integer value of `1` if the logical expression evaluates to `true`
  - Returns an integer value of `0` otherwise



```
cout << ('a' == 97);
cout << ('a' > 'z');
cout << (97.0 == 97);
```

```
101
```

Relational Operators and the `string` Type

- Relational operators can be applied to strings
- Strings are compared character by character, starting with the first character
- Comparison continues until either a mismatch is found or all characters are found equal
- If two strings of different lengths are compared and the comparison is equal to the last character of the shorter string
  - The shorter string is less than the larger string

Relational Operators and the `string` Type

```
“Hello” < “Hi”            true
“Hello” > “Hen”           false
“Air” < “An”              true
“Hello” == “hello”        false
“Air” <= “Bill”           true
“Hi” > “Bill”             true
“Hello” > “hello”         false
“Bill” >= “Billy”         false
“Big” <= “Bigger”         true
```

Logical Operators and Logical Expressions

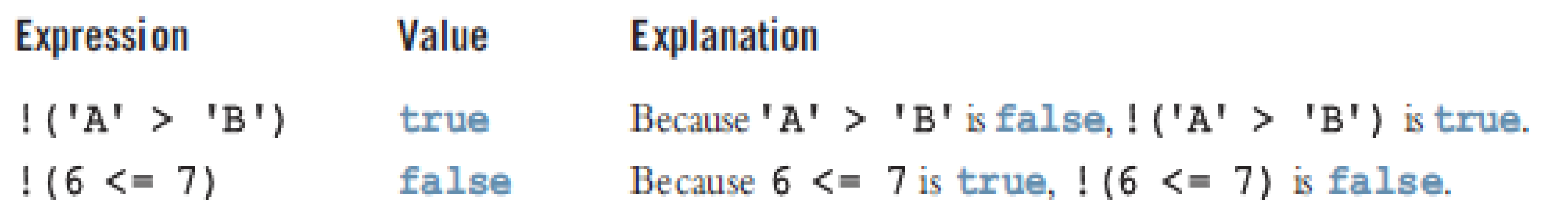
- **Logical (Boolean) operators** enable you to combine logical expressions

Operator	Description
!	not
&&	and
||	or

Logical Operators and Logical Expressions

- The negation (!) operator

Expression	!Expression
true (nonzero)	false (0)
false (0)	true (1)



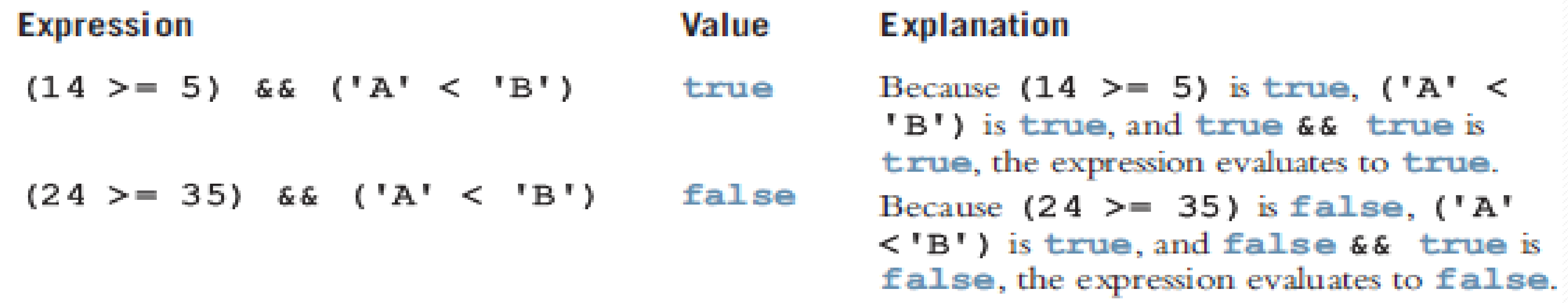
Expression	Value	Explanation
`!('A' > 'B')`	`true`	Because `'A' > 'B'` is `false`, `!('A' > 'B')` is `true`.
`!(6 <= 7)`	`false`	Because `6 <= 7` is `true`, `!(6 <= 7)` is `false`.

Logical Operators and Logical Expressions

- The AND (&&) Operator

Expression1	Expression2	Expression1 && Expression2
true (nonzero)	true (nonzero)	true (1)
true (nonzero)	false (0)	false (0)
false (0)	true (nonzero)	false (0)
false (0)	false (0)	false (0)

false && anything -> false



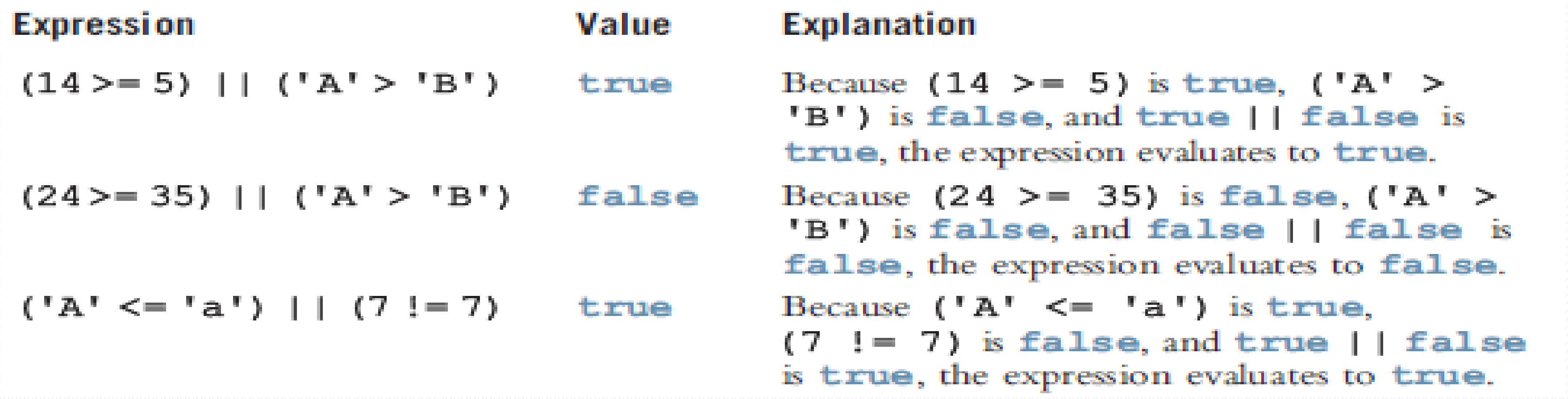
Expression	Value	Explanation
`(14 >= 5) && ('A' < 'B')`	`true`	Because `(14 >= 5)` is `true`, `('A' < 'B')` is `true`, and `true && true` is `true`, the expression evaluates to `true`.
`(24 >= 35) && ('A' < 'B')`	`false`	Because `(24 >= 35)` is `false`, `('A' < 'B')` is `true`, and `false && true` is `false`, the expression evaluates to `false`.

Logical Operators and Logical Expressions

- The OR (||) Operator

true || anything -> true

Expression1	Expression2	Expression1 || Expression2
true (nonzero)	true (nonzero)	true (1)
true (nonzero)	false (0)	true (1)
false (0)	true (nonzero)	true (1)
false (0)	false (0)	false (0)



Expression	Value	Explanation
`(14 >= 5) || ('A' > 'B')`	`true`	Because `(14 >= 5)` is `true`, `('A' > 'B')` is `false`, and `true || false` is `true`, the expression evaluates to `true`.
`(24 >= 35) || ('A' > 'B')`	`false`	Because `(24 >= 35)` is `false`, `('A' > 'B')` is `false`, and `false || false` is `false`, the expression evaluates to `false`.
`('A' <= 'a') || (7 != 7)`	`true`	Because `('A' <= 'a')` is `true`, `(7 != 7)` is `false`, and `true || false` is `true`, the expression evaluates to `true`.

Logical Operators and Logical Expressions

- **Properties of Boolean algebra**

Operator	Formula
Commutativity	A && B = B && A A || B = B || A
Associativity	(A && B) && C = A && (B && C) (A || B) || C = A || (B || C)
Distributivity	A && (B || C) = (A && B) || (A && C) A || (B && C) = (A || B) && (A || C)
Double Negation	! (! A) = A
De Morgan’s Law	! (A && B) = (! A) || (! B) ! (A || B) = (! A) && (! B)

Order of Precedence

- Relational and logical operators are evaluated from left to right
- The associativity is left to right
- Parentheses can override precedence

Order of Precedence

1. + (positive) , - (negative), ! (not)
2. ++ (increment), -- (decrement)
3. * (multiplication), / (division), % (modulus)
4. + (addition), - (subtraction)
5. <, <=, >, >= (relational comparison)
6. == (equal-to), != (not-equal-to)
7. && (and)
8. || (or)
9. = (assignment), +=, -=, *=, /=, %= (compound assignment)

Order of Precedence

$$x=9 \;\&\&\; 8 == 7 < \underline{!6} * \underline{-5} \,/\, 4 + 3 \,\%\, 2 - 1$$

$$x=9 \;\&\&\; 8 == 7 < \underline{0 * (-5)} \,/\, 4 + \underline{3 \,\%\, 2} - 1$$

$$x=9 \;\&\&\; 8 == 7 < \underline{0 \,/\, 4} + 1 - 1$$

$$x=9 \;\&\&\; 8 == 7 < \underline{0 + 1 - 1}$$

$$x=9 \;\&\&\; 8 == \underline{7 < 0}$$

$$x=9 \;\&\&\; \underline{8 == 0}$$

$$x=\underline{9 \;\&\&\; 0}$$

$$\underline{x= 0}$$

`int` and bool Data Type and Logical Expressions

- Logical expressions evaluate to either `true` (=`1`) or `false` (=`0`)
- The data type **`bool`** has logical (Boolean) values `true` and `false`

```
    bool legalAge = (age >= 21);
```

- Earlier versions of C++ did not provide built-in data types that had Boolean values, so the **`int`** data type was used to manipulate logical (Boolean) expressions

```
  int legalAge = (age >= 21);
```

Selection: `if` and `if...else`

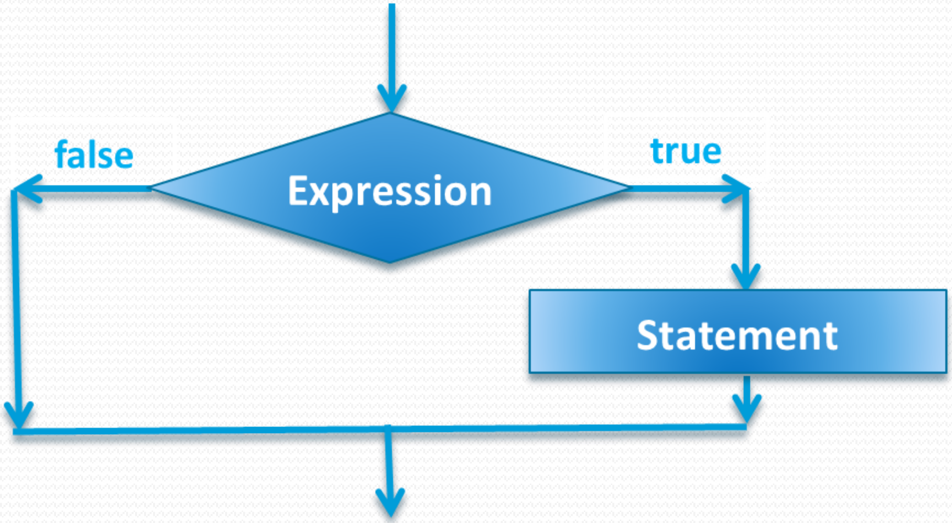
- One-Way Selection
- Two-Way Selection
- Compound (Block of) Statements
- Multiple Selections: Nested `if`
- Comparing `if...else` Statements with a Series of `if` Statements
- Multi-Way Selection

One-Way Selection

- The syntax is:

```
if (Expression)
    Statement
```

  - The Statement is executed if the value of the Expression is `true`
  - The Statement is bypassed if the value is `false` and the program goes to the next statement



One-Way Selection

- Examples:

```
grade = ‘F’;
if (score >= 60)
   grade = ‘P’;
```

CORRECT – but we do need the `grade = ‘F’;` assignment before the if to make sure the grade will have a value if the score is lower than 60

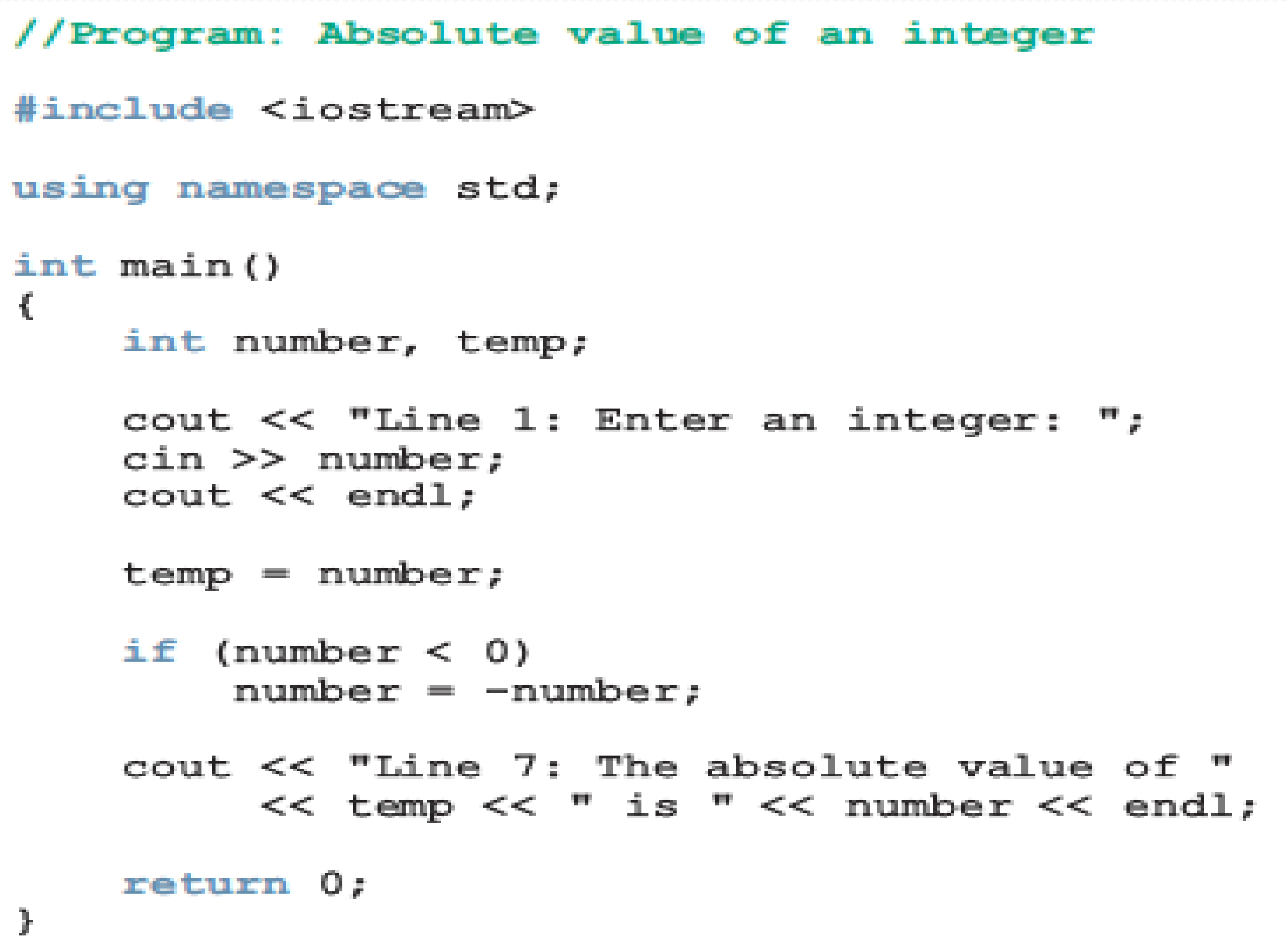
```
grade = ‘F’;
if score >= 60
   grade = ‘P’;
```

INCORRECT – missing parentheses

```
grade = ‘F’;
if (score >= 60);
   grade = ‘P’;
```

SYNTACTICALLY CORRECT – it has a valid if with an empty statement – that does not do (`if (score >= 60);`) , followed by an assignment `grade = ‘P’` but **LOGICALLY INCORRECT** – it will make the `grade = ‘P’` for any score so you need to delete the ; at the end of the first line

One-Way Selection

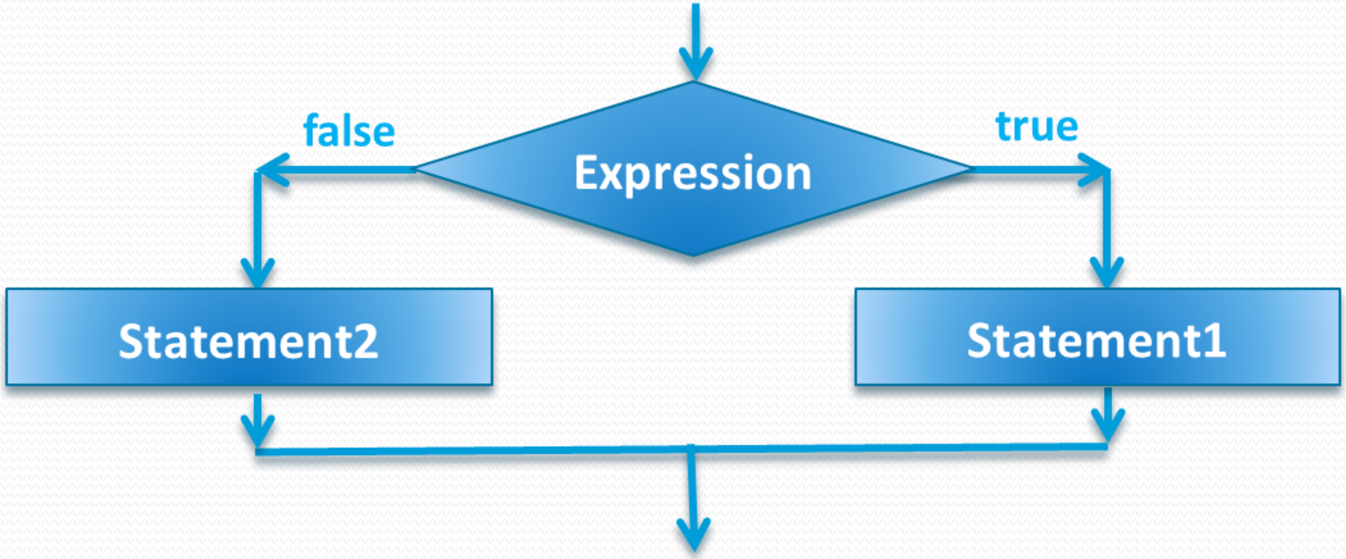


```
//Program: Absolute value of an integer

#include <iostream>

using namespace std;

int main()
{
    int number, temp;

    cout << "Line 1: Enter an integer: ";
    cin >> number;
    cout << endl;

    temp = number;

    if (number < 0)
        number = -number;

    cout << "Line 7: The absolute value of "
         << temp << " is " << number << endl;

    return 0;
}
```

Two-Way Selection

```
if  (Expression)
   Statement1
else
   Statement2
```

- If expression is `true`, `Statement1` is executed; otherwise, `Statement2` is executed



Two-Way Selection

- Examples:

```
if (hours > 40.0)
    wages = 40.0 * rate + 1.5 * rate * (hours – 40.0);
else
    wages = hours * rate;

if (score >= 60)
    grade = ‘P’;
else
    grade = ‘F’;
```

Two-Way Selection

- What is wrong with this code?

One-way if

```
if (Expression);
```

```
    Statement1;
```

Statement after if

```
else
```

else without an if

```
    Statement2;
```

Syntactic error

```
if (Expression)
    Statement1;
else ;
```

two-way if

```
    Statement2;
```

Statement after if

Logical error

Two-Way Selection

- What is wrong with this code?

One-way if

```
if (Expression)
    Statement1;
    Statement3;
else
    Statement2;
```

Statement after if

else without an if

Syntactic error

two-way if

```
if (Expression)
    Statement1;
else
    Statement2;
    Statement3;
```

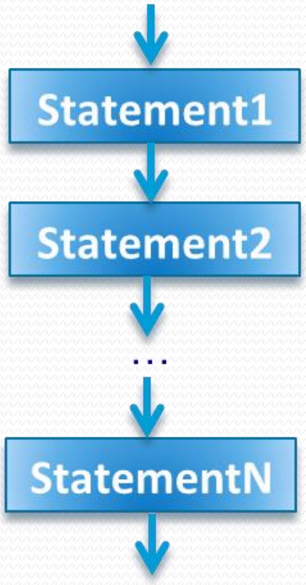
Statement after if

Logical error

Compound (Block of) Statements

- **Compound statement** (block of statements):

```
{
    Statement1;
    Statement2;
    …
    StatementN;
}
```



- A compound statement is a single statement
- No need for a semi-colon at the end

Parentheses/Brackets

Parentheses/Brackets	Name
()	Parentheses
{}	Curly brackets
[]	Square brackets
<>	Angle brackets

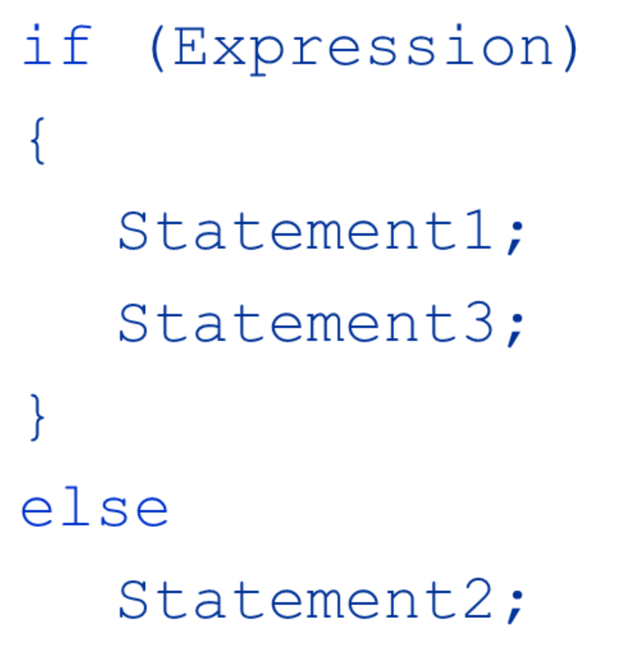
Compound (Block of) Statements

- Example:

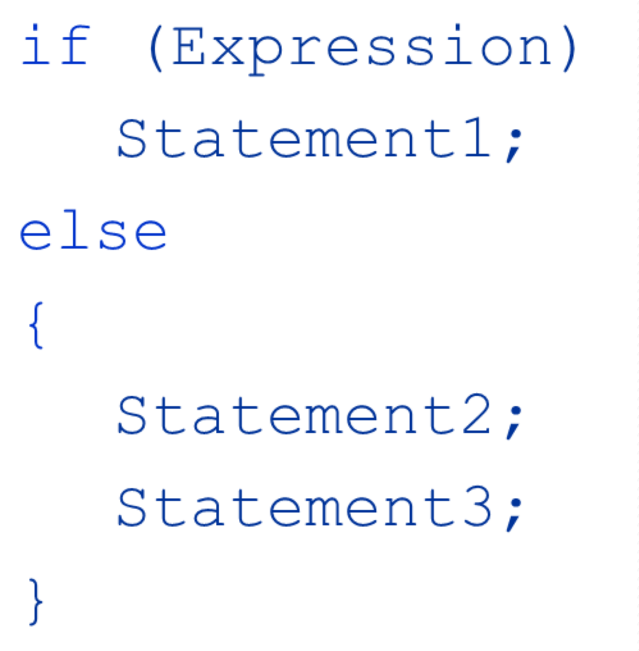
```
if (age > 18)
{
  cout << "Eligible to vote." << endl;
  cout << "No longer a minor." << endl;
}
else
{
  cout << "Not eligible to vote." << endl;
  cout << "Still a minor." << endl;
}
```

Compound (Block of) Statements

- If multiple statements in a branch, use compound statement



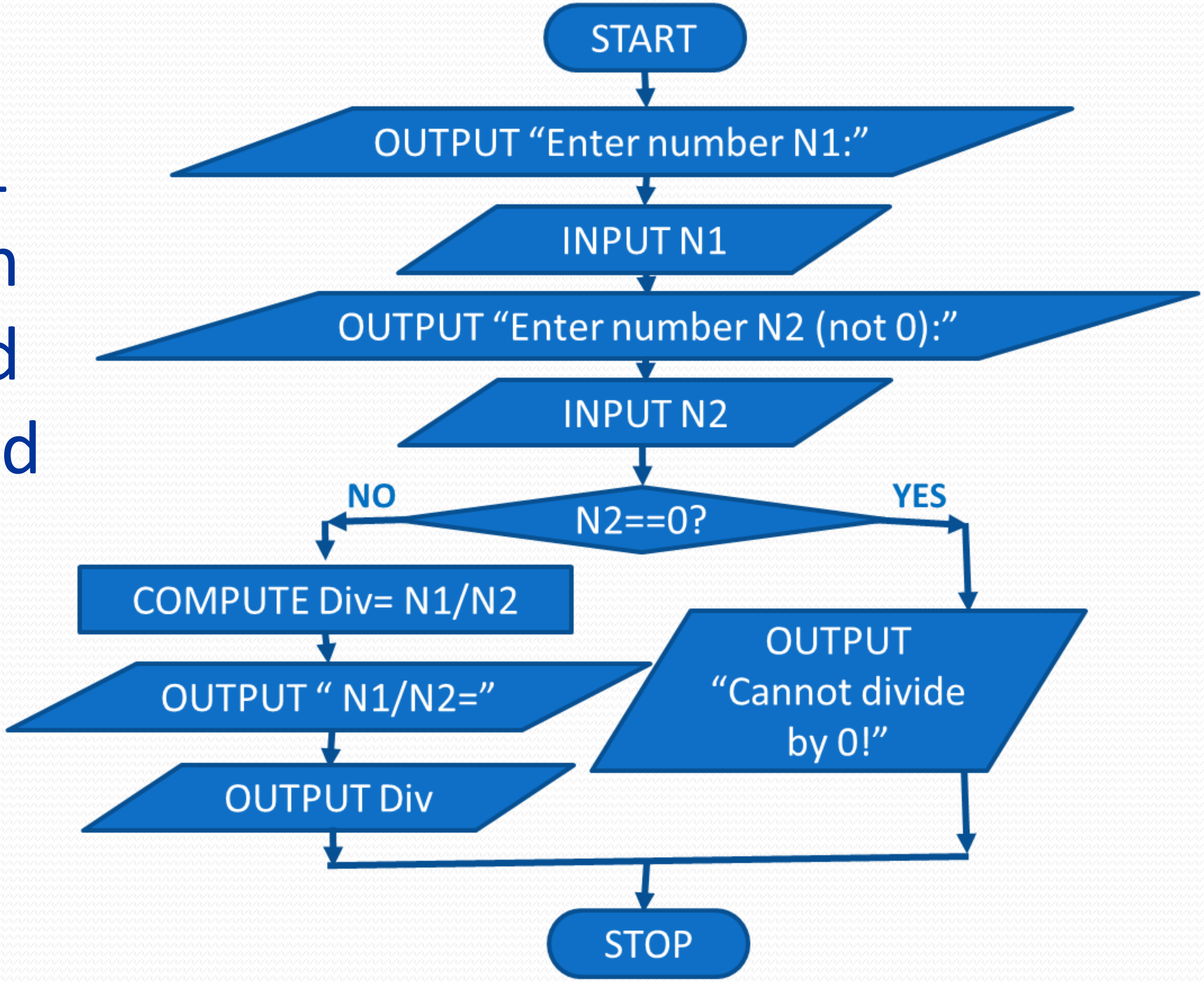
```
if (Expression)
{
   Statement1;
   Statement3;
}
else
   Statement2;
```



```
if (Expression)
   Statement1;
else
{
   Statement2;
   Statement3;
}
```

Exercise: Division of 2 Numbers

- Read 2 numbers N1 and N2 from the user and compute and output the division/ quotient of the 2 numbers


START
OUTPUT “Enter number N1:”
INPUT N1
OUTPUT “Enter number N2 (not 0):”
INPUT N2
NO
N2==0?
YES
COMPUTE Div= N1/N2
OUTPUT “ N1/N2=”
OUTPUT Div
OUTPUT “Cannot divide by 0!”
STOP

Exercise: Division of 2 Numbers

```
cout << "This program read 2 integral numbers from the user, "
<< "divides them, and outputs the quotient\n\n";

//INPUT N1
//declare variable
int N1;
//prompt the user for a number
cout << "Enter a number N1: ";
//read the number
cin >> N1;
//output the number
cout << "N1=" << N1;

//INPUT N2
//declare variable
int N2;
//prompt the user for a number
cout << "\n\nEnter a number N2: ";
//read the number
cin >> N2;
//output the number
cout << "N2=" << N2;
```

Exercise: Division of 2 Numbers

```
//check if N2 is not 0
if (N2 != 0)
{
     //COMPUTE Div=N1/N2
     //declare variable
     float Div;
     //assign value to it
     Div = static_cast<float>(N1) / N2;

     //OUTPUT Div
     cout << fixed << showpoint << setprecision(2);
     cout << "\n\n" << N1 << " divided by " << N2 << " is " << Div;
}
else
cout << "\n\nCannot divide by 0!";

//STOP

//prevent the console from closing
cout << "\n\nPress any key to exit";
_getch();
```

Exercise: Minimum of 2 Numbers

- Compute and output the minimum/smallest of the 2 numbers N1 and N2

```
//Compute the minimum/smallest of N1 and N2 and output it
//declare variable for minimum
int Min;
//compare N1 and N2
if (N1 < N2)
       Min = N1;
else //N1>=N2
       Min = N2;
//output the minimum
cout << "\n\nThe minimum of " <<N1<< " and " <<N2<< " is " <<Min;
```

Exercise: Maximum of 2 Numbers

- Compute and output the maximum/largest of 2integral numbers N1 and N2

```
//declare variable for maximum
int Max;
//compare N1 and N2
if (N1 > N2)
       Max = N1;
else
       Max = N2;
cout << "\n\nThe maximum of " <<N1<< " and " <<N2<< " is " << Max;
```

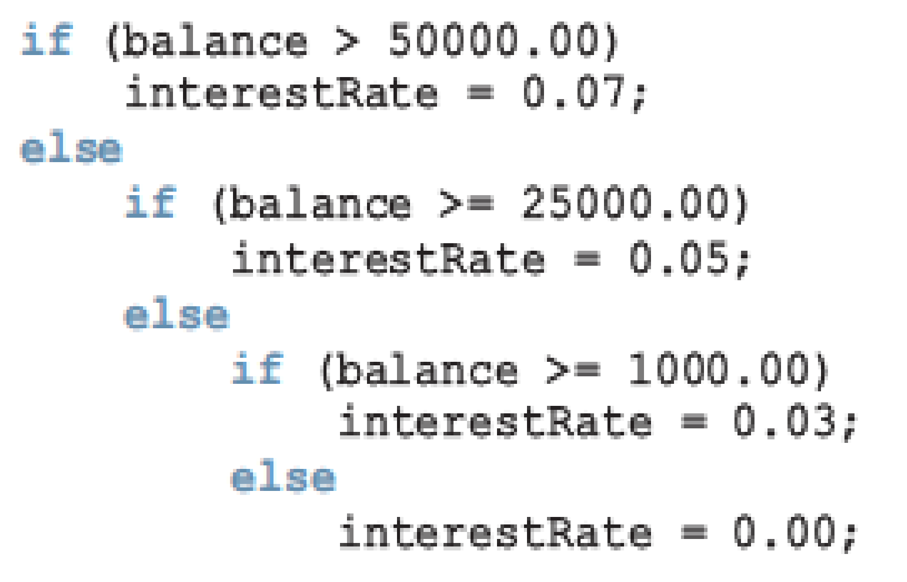
Multiple Selections: Nested `if`

- **Nesting**: one control statement in another
- An `else` is associated with the most recent `if` that has not been paired with an `else`
- You can nest control statement in compound statements to remove some of the ambiguity or organize your code

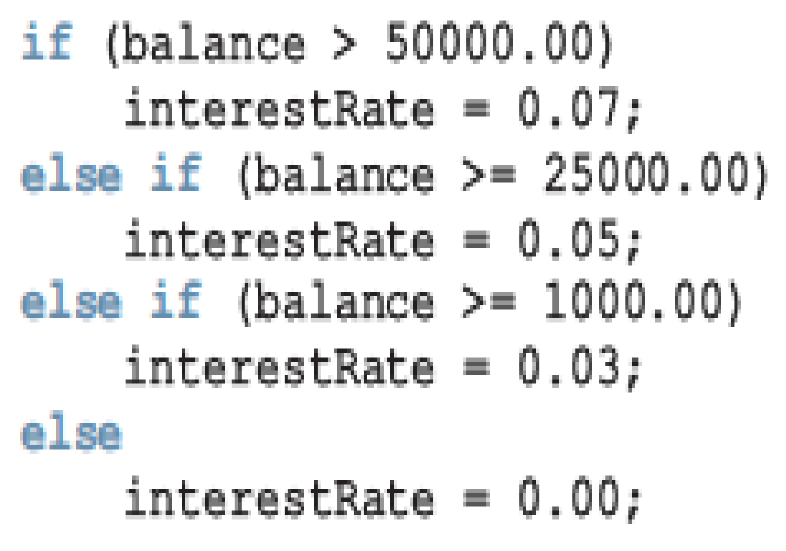
```
if (expression1)
    statement1;
else
   if (expression2)
      statement2;
   else
       statement3;
```

```
if (expression1)
    statement1;
else
{
   if (expression2)
      statement2;
   else
       statement3;
}
```

Multiple Selections: Nested `if`

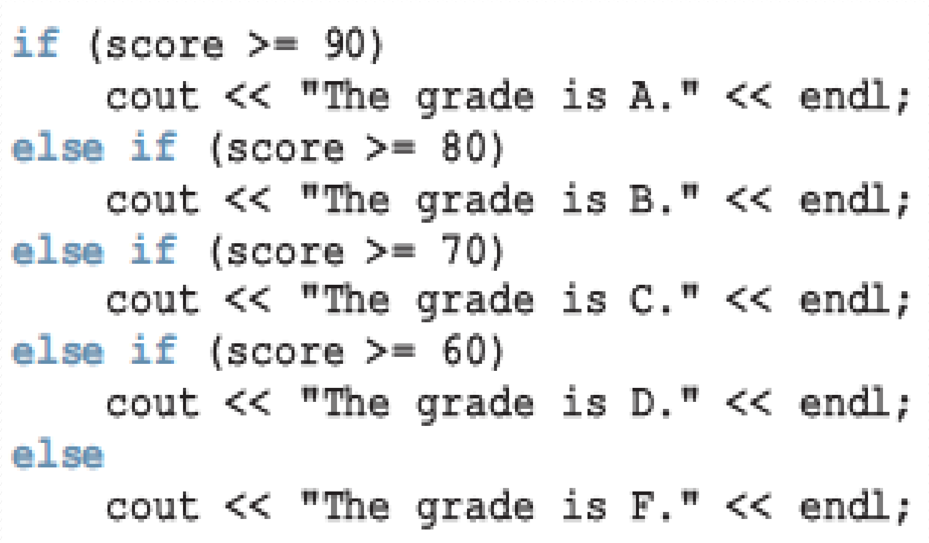


```
if (balance > 50000.00)
    interestRate = 0.07;
else
    if (balance >= 25000.00)
        interestRate = 0.05;
    else
        if (balance >= 1000.00)
            interestRate = 0.03;
        else
            interestRate = 0.00;
```



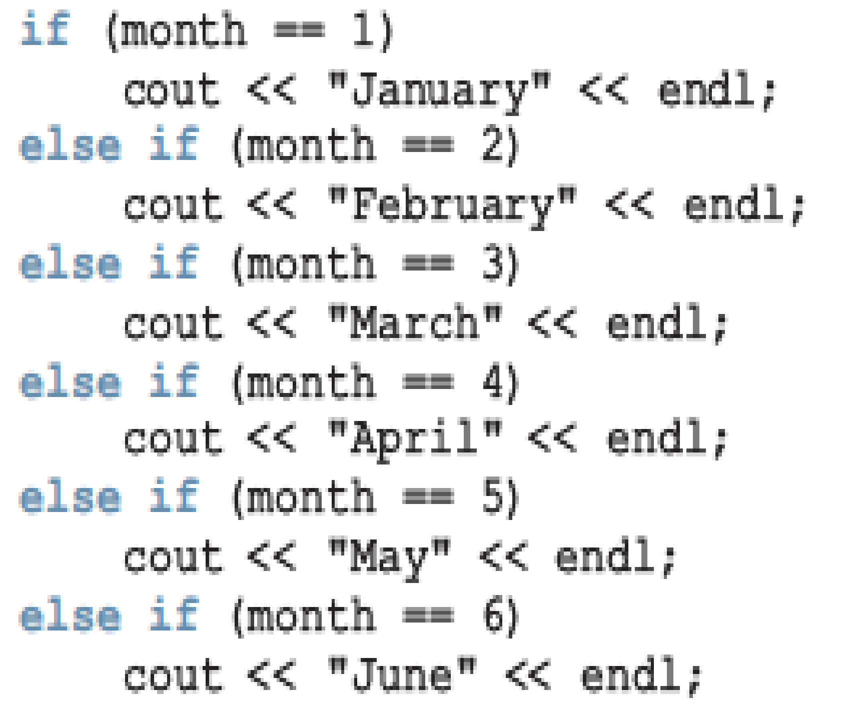
```
if (balance > 50000.00)
    interestRate = 0.07;
else if (balance >= 25000.00)
    interestRate = 0.05;
else if (balance >= 1000.00)
    interestRate = 0.03;
else
    interestRate = 0.00;
```

Multiple Selections: Nested `if`

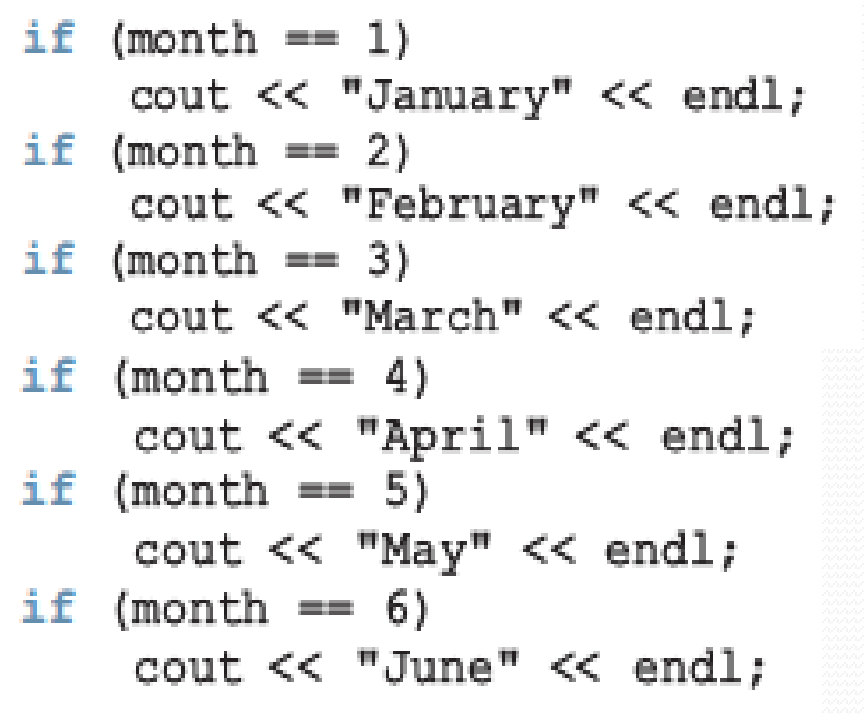


```
if (score >= 90)
    cout << "The grade is A." << endl;
else if (score >= 80)
    cout << "The grade is B." << endl;
else if (score >= 70)
    cout << "The grade is C." << endl;
else if (score >= 60)
    cout << "The grade is D." << endl;
else
    cout << "The grade is F." << endl;
```

Comparing if…else Statements with a Series of if Statements



```
if (month == 1)
    cout << "January" << endl;
else if (month == 2)
    cout << "February" << endl;
else if (month == 3)
    cout << "March" << endl;
else if (month == 4)
    cout << "April" << endl;
else if (month == 5)
    cout << "May" << endl;
else if (month == 6)
    cout << "June" << endl;
```



```
if (month == 1)
    cout << "January" << endl;
if (month == 2)
    cout << "February" << endl;
if (month == 3)
    cout << "March" << endl;
if (month == 4)
    cout << "April" << endl;
if (month == 5)
    cout << "May" << endl;
if (month == 6)
    cout << "June" << endl;
```

Exercise: Letter Grade Using Nested Ifs

- Compute the letter grade (A, B, C, D, F) from the numeric grade

Numeric Grade	Letter Grade
90 – 1000	**A**
80 – 89.99	**B**
70 – 79.99	**C**
60 – 69.99	**D**
0 – 59.99	**F**

Exercise: Letter Grade Using Nested Ifs

```
// starting with the lowest grade
char LetterGrade;
if (NumericGrade < 60)
        LetterGrade = 'F';
else //above 60
        if (NumericGrade < 70)
                LetterGrade = 'D’;
        else //above 70
                if (NumericGrade < 80)
                        LetterGrade = 'C’;
                else //above 80
                        if (NumericGrade < 90)
                                LetterGrade = 'B’;
                        else //above 90
                                LetterGrade = 'A';
cout << "\nThe letter grade is " << LetterGrade << "\n";
```

Exercise: Letter Grade Using Nested Ifs

```
//OR
// starting with the highest grade
if (NumericGrade >= 90)
        LetterGrade = 'A';
else if (NumericGrade >= 80)
        LetterGrade = 'B';
else if (NumericGrade >= 70)
        LetterGrade = 'C';
else if (NumericGrade >= 60)
        LetterGrade = 'D';
else
        LetterGrade = 'F';
cout << "\nThe letter grade is " << LetterGrade << "\n";
```

Short-Circuit Evaluation

- **Short-circuit evaluation**: evaluation of a logical expression stops as soon as the value of the expression is known
  - `true || anything -> true`
  - `false && anything -> false`
- Example:

```
(age >= 21) || ( x == 5)
(grade == 'A') && (x >= 7)
```

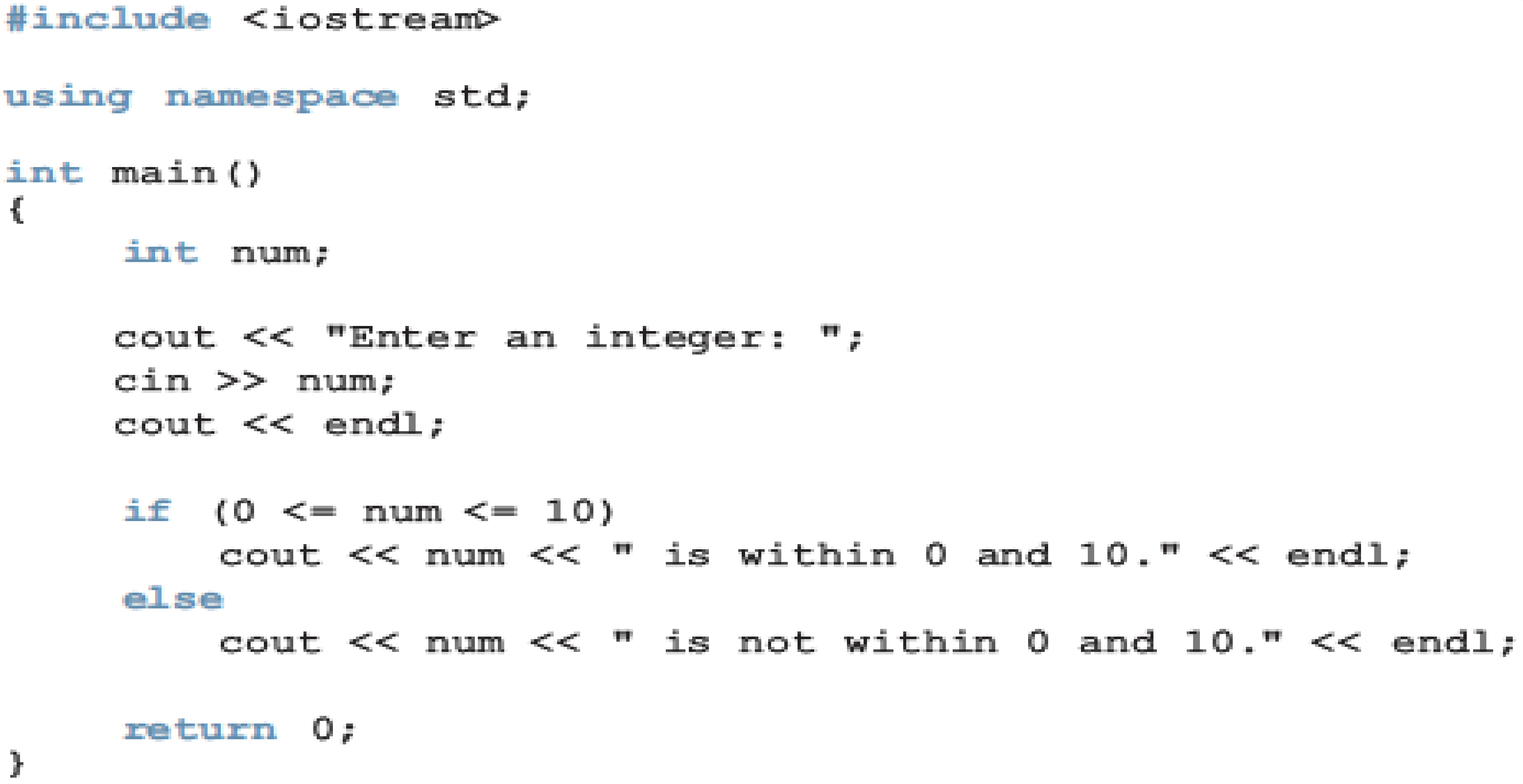
Comparing Floating-Point Numbers for Equality: A Precaution

- Comparison of floating-point numbers for equality may not behave as you would expect
- Math: $\frac{3}{7}+\frac{2}{7}+\frac{2}{7}=\frac{3+2+2}{7}=\frac{7}{7}=1$
- C++:
  - int: `3/7 + 2/7 + 2/7 = 0 !=1`
  - float:

3.0/7.0 + 2.0/7.0 + 2.0/7.0 =
0.42857142~~857~~+0.2857142~~8571~~+0.2857142~~8571~~ = 0.9999994
!=1

- Use a tolerance value `fabs(x - y) < 0.000001`

Associativity of Relational Operators: A Precaution



```cpp
#include <iostream>

using namespace std;

int main()
{
    int num;

    cout << "Enter an integer: ";
    cin >> num;
    cout << endl;

    if (0 <= num <= 10)
        cout << num << " is within 0 and 10." << endl;
    else
        cout << num << " is not within 0 and 10." << endl;

    return 0;
}
```

Associativity of Relational Operators: A Precaution

- 0<=num<=10

0 10

-5	5	15
0<=-5 <=10	0<=5 <=10	0<=15<=10
0 <=10	1 <=10	1 <=10
1	1	1

Always `true`

- 0<=num && num<=10

0 10

-5	5	15
0<=-5 && -5<=10	0<=5 && 5<=10	0<=15 && 15<=10
0 && 1	1 && 1	1 && 0
0	1	0

`true` only if between 0 and 10

Avoiding Bugs by Avoiding Partially Understood Concepts and Techniques

- Must use concepts and techniques correctly;
  - Otherwise solution will be either incorrect or deficient
- If you do not understand a concept or technique completely
  - Don’t use it
  - Save yourself an enormous amount of debugging time

Input Failure and the `if` Statement

- If input stream enters a fail state
  - All subsequent input statements associated with that stream are ignored
  - Program continues to execute
  - May produce erroneous results
- Can use `if` statements to check status of input stream
- If stream enters the fail state, include instructions that stop program execution

Confusion Between the Equality (==) and Assignment (=) Operators

- C++ allows you to use any expression that can be evaluated to either `true` or `false` as an expression in the `if` statement: `(x=5)` or `(x==5)`
- The appearance of '=' in place of '==' resembles a *silent killer*
  - It is not a syntax error, it is a logical error

```
if (x = 5)
    cout << "The value is five." << endl;
else
    cout << "The value is five." << endl;
```

Conditional Operator (?:)

- **Conditional operator (? :)** takes three arguments (ternary operator)
- Syntax for using the conditional operator:

```
expression1 ? expression2 : expression3
```

- If `expression1` is `true`, the result of the conditional expression is `expression2`
  - Otherwise, the result is `expression3`

```
(a >= b) ? a : b;
```

Conditional Operator (?:)

- With conditional operator:

```
variable = expression1 ? expression2 : expression3
```

- With if…else statement:

```
if (expression1)
  variable = expression2;
else
  variable = expression3;
```

- Example:

```
  max = (a >= b) ? a : b;
```

Conditional Operator (?:)

- With conditional operator:

```
cout << ( expression1 ? expression2 : expression3 );
```

- With if…else statement:

```
if (expression1)
  cout << expression2;
else
  cout << expression3;
```

- Example:

```
  cout << ( (a >= b) ? a : b );
```

Conditional Operator (?:)

- With if-else:

```
if (score >= 60)
    grade = ‘P’;
else
    grade = ‘F’;
```

- With conditional statement:

```
grade=(score >= 60) ? ‘P’:‘F’;
```

Exercise: Maximum and Minimum Using Conditional Operator

- Compute and output the minim and maximum of 2 integral numbers N1 and N2 using the conditional operator

```
//Maximum of N1 and N2 using conditional operator
Max = (N1 > N2) ? N1 : N2;
cout << "\n\nThe maximum of " << N1 << " and " << N2 << " is " << Max;

/Minimum of N1 and N2
cout << "\n\nThe minimum of " << N1 << " and " << N2 << " is "
               << ((N1 < N2) ? N1 : N2);
```

Program Style and Form (Revisited): Indentation

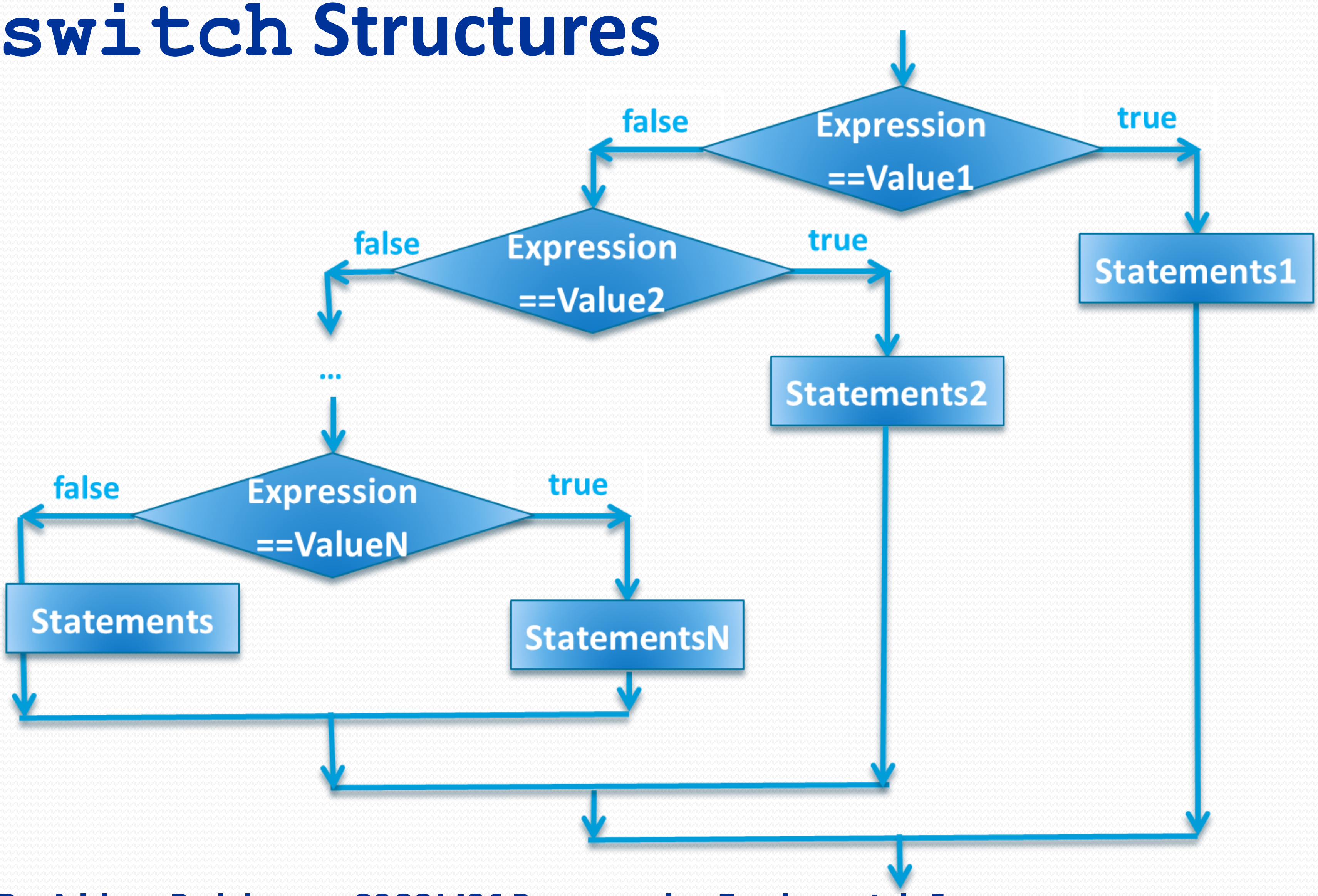
- If your program is properly indented
  - Spot and fix errors quickly
  - Show the natural grouping of statements
- Insert a blank line between statements that are naturally separate
- Two commonly used styles for placing braces
  - On a line by themselves
  - Or left brace is placed after the expression, and the right brace is on a line by itself

`switch` Structures

- **`switch` structure**: alternate to if-else
- `switch` (integral) expression is evaluated first
- Value of the expression determines which corresponding action is taken
- Expression is sometimes called the selector

```
switch (Expression)
{
   case Value1:
        Statements1
        break;
   case Value2:
        Statements2
        break;
   …
   case ValueN:
        StatementsN
        break;
   default:
        Statements
}
```

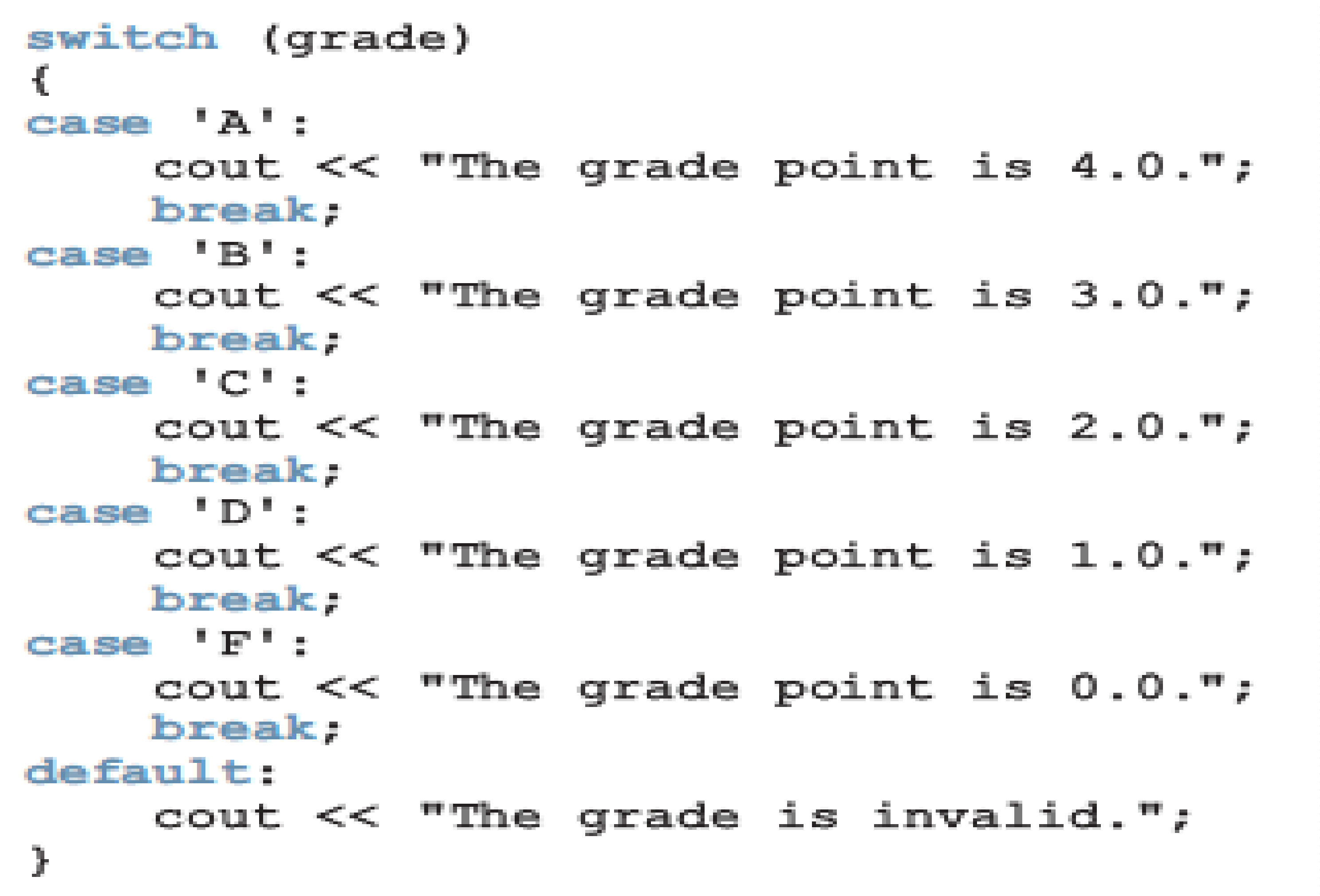
`switch` Structures



`switch` Structures

- One or more statements may follow a case label
- Braces are not needed to turn multiple statements into a single compound statement
- The `break` statement may or may not appear after each statement
- `switch`, `case`, `break`, and `default` are reserved words

switch Structures



```
switch (grade)
{
case 'A':
      cout << "The grade point is 4.0.";
      break;
case 'B':
      cout << "The grade point is 3.0.";
      break;
case 'C':
      cout << "The grade point is 2.0.";
      break;
case 'D':
      cout << "The grade point is 1.0.";
      break;
case 'F':
      cout << "The grade point is 0.0.";
      break;
default:
      cout << "The grade is invalid.";
}
```

Exercise: Grade Points

- Compute the Grade Points from the Letter Grade

```
double GradePoints;
switch (LetterGrade)
{
case 'A':
    GradePoints = 4;
    break;
case 'B':
    GradePoints = 3;
    break;
case 'C':
    GradePoints = 2;
    break;
case 'D':
    GradePoints = 1;
    break;
default:
    cout << "\n" << LetterGrade<<" is invalid!";
    GradePoints = 0;
}
cout<<"\nThe grade points value is "<<GradePoints;
```

Avoiding Bugs by Avoiding Partially Understood Concepts and Techniques: Revisited

- A missing `break` statement will cause the next statement to be called (even if the `case value` is met)
- To output results correctly
  - The `switch` structure must include a `break` statement after each `cout` statement

Terminating a Program with the `assert` Function

- Certain types of errors that are very difficult to catch can occur in a program
    - Example: division by zero can be difficult to catch using any of the programming techniques examined so far

```
quotient=numerator / denominator;
```

The `assert` Function

- The predefined function, **`assert`**, is useful in stopping program execution when certain elusive errors occur
- Syntax:

```
assert (Expression);
```

  - `Expression` is any logical expression
  - If `Expression` evaluates to `true`, the next statement executes
  - If `Expression` evaluates to `false`, the program terminates and indicates where in the program the error occurred
  - To use `assert`, include `cassert` header file

The `assert` Function

- `assert` is useful for enforcing programming constraints during program development
- After developing and testing a program, remove or disable assert statements
- Place the preprocessor directive `#define NDEBUG` before the directive `#include <cassert>`

```
#define NDEBUG
#include <cassert>
assert(denominator); //stops if denominator is 0
quotient = numerator / denominator;
```

Summary

- Control structures
- Relational operators
- Logical expressions
- Logical operators
- Selection structures
- If, if-else, and switch statement
- Compound statement
- assert statement