Programming in C/C++

Lecture 4: Selection Statements and Random Numbers

Kristina Shroyer

Objectives

You should be able to describe:

- Relational Expressions
- The **`if-else`** Statement
- Nested **`if`** Statements
- **`if-else if`** statement chains
- The **`switch`** Statement
- Simple/Pseudo Random Number Generators
- Common Programming Errors

Altering the Normal Flow of Control – Selection and Repetition

- **Flow of Control**: the order in which a program's statements are executed
  - Normal flow is sequential
- Selection and Repetition Statements allow programmer to alter normal sequential flow of statement execution in a program
- **Selection**: makes it possible to select a particular statement to be executed next
  - Selection choice is made from a well-defined set
  - Selection involves choosing between alternatives
- **Repetition**: allows a set of statements to be repeated

Relational Operands/Operators

- **Concept:** Relational Operators allow you compare numeric values and determine if one is greater than, less than, equal to or not equal to the other

- All computers are able to compare numbers
  - **Numeric data** in C++ are compared using relational operators
  - **Characters** and **bools** can also be compared in C++ using these relational operators because characters and bools are considered numeric values in C++
  - Each relational operator determines a specific relationship between two values

- All relational operators are **binary operators** with **left to right associativity**

TABLE 4.1 *Relational Operators in C++*

Relational Operator	Meaning	Example
<	less than	`age < 30`
>	greater than	`height > 6.2`
<=	less than or equal to	`taxable <= 20000`
>=	greater than or equal to	`temp >= 98.6`
==	equal to	`grade == 100`
!=	not equal to	`number != 250`

Relational Expressions

- **Relational Expressions**: expressions that use relational operators to compare operands
- **Format for a simple relational expression**: a relational operator connecting two variable and/or constant operands
- Relational operators may be used with integer, boolean or character data
  - What makes it possible to compare characters?
- Examples of valid relational expressions:

```
age > 40   length <= 50    flag == done
```

Relational Expressions

- **Relational Expressions** (also called conditions):
  - As with all expressions in C++, relational expressions are evaluated to yield a numerical result
    - A Condition that is **true** evaluates to 1
    - A Condition that is **false** evaluates to 0
  - Also remember a `bool` is evaluated behind the scenes numerically in C++: meaning anything that evaluates to zero is false, ***anything that evaluates to anything other than zero is true***
    - ***This is important and can cause issues in your code***

- **Example**:
  - The relationship `2.0 > 3.3` is always false, therefore the expression has a value of `0`
  - **Example code: (RelationalExp.cpp – Example 1**)

```
#include <iostream>
using namespace std;

int main()
{
 int x = 5;

 cout << "The value of x < 10 is " << (x < 10) << endl;
 cout << "The value of x > 10 is " << (x > 10) << endl;

 system("PAUSE");
 return 0;
}
```

Relational Expressions

- **Example**: **(RelationalExp2.cpp – Example 2**)
  - Another example illustrating that relational expressions evaluate to 1 if they are true and 0 if they are false

```
#include <iostream>
using namespace std;

int main()
{
 //declare and initialize variables
 bool trueValue;
 bool falseValue;
 int x = 5;
 int y = 10;

 trueValue  = x < y;
 falseValue = y == x;

 cout << "True is " << trueValue << endl;
 cout << "False is " << falseValue << endl;

 system("PAUSE");
 return 0;
}
```

  - Let's look at these two statements: `trueValue  = x < y;` `falseValue = y == x;`
  - These two statements assign the resulting value of a relational expression to a variable of type bool (remember bool is represented in C++ as 0 for false and any other number for true)
  - In both cases the relational operation (< , ==) was carried out before the assignment operation was performed.
    - ***Relational Operators have a higher precedence than the assignment operator***

Relational Expressions - Precedence

- Relational operators have higher precedence than the assignment operator (see last example)
- ***Arithmetic Operators have higher precedence than Relational Operators which have higher precedence than the assignment operator (the = has very low precedence which should make sense)***

- **Example: (RelationalExp3.cpp – Example 3)**

```
#include <iostream>
using namespace std;

int main()
{
 //declare and initialize variables
 int x = 10;
 int y = 7;
 bool z;

 z = x < y;
 cout << "\nIn (z = x < y;) z is " << z
      << " because x is NOT less than y" << endl;

 //Notice the parenthesis, prints 1 for true
 cout << "\n(x > y) evaluates to " << (x > y) << endl;

 //notice the precedence of the addition operator over the relational op
 cout << "\n( x == y + 3 ) evaluates to " << (x == y + 3) << endl;

 //Notice the precedence of the relational op over the assignment op
 cout << "\n(z = y != x) is " << (z = y !=x) << endl;

 system("PAUSE");
 return 0;
}
```

Relational Expressions - Precedence

- Precedence of Relational Operators in relation to each other (highest to lowest)
  - For operators with the same precedence, left to right associativity applies
- Example: (Relational4.cpp – Example 4)

```
#include <iostream>
using namespace std;

int main()
{
 //declare and initialize variables
 int a = 9;
 int b = 24;
 int c = 0;

 cout << "(c == a > b) is " << (c == a > b) << endl;

 system("PAUSE");
 return 0;
}
```

>, =>, <, <=
== !=

Relational Expressions

- **Results of Relational Expressions**:
  - The relational expression's evaluated value (0 or 1) is not as important in C++ as the interpretation C++ places on the value when it is used in a selection statement
    - **In the selection statements in C++, 0 is used to represent a false condition and any number other than 0 is used to represent a true condition**

- Comparing characters using relational expressions
  - When comparing letters such as 'A' and 'B' we compare the characters ASCII values
  - it is important to realize that each alphabetical letter that comes before another in the alphabet is stored using a lower numerical value than the later letter (so 'A' is < 'B') (this is due to the way the ASCII code is set up – See Appendix B)
    - This is true for both lowercase and uppercase letters
  - However, notice that the lowercase letters have higher ASCII codes than the uppercase
    - 'a' is represented by 97 and 'A' is represented by 65
    - So ('a' > 'A') evaluates to true

Logical Operators

- **Concept:** Logical operators connect two or more relational expressions into one or reverse the logic of an expression

- There are three logical operators:
  - AND and OR are connecting logical operators
  - NOT reverses the logic of an expression

Operator	Meaning
&&	AND
||	OR
!	NOT

Logical Operators

- **`AND` Operator, `&&`:**
  - Connects two relational expressions into one
    - Both expressions must be true for the overall expression to be true
  - **Example**: `(age > 40) && (term < 10)`
    - Compound condition is true (has value of 1) only if `age > 40` and `term < 10`
    - Note that the parenthesis could have been omitted in this expression this is true **because relational operators have higher precedence than logical operators**

- Notice with an && logical operator if Expression 1 is false the result of the && will always be false. Therefore the compiler does not need to check Expression 2 if Expression 1 is false. This is called **short circuit evaluation**.

Expression 1	Expression 2	Expression 1 && Expression 2
false	false	false (0)
false	true	false (0)
true	false	false (0)
true	true	true (1)

Logical Operators

- **`OR` Operator, `||`:**
  - Connects two expressions into one. In order for the entire expression to be true either one OR both statements must be true. It is only necessary for one to be true it doesn't matter which.
    - There is no || key on the key board, use two | symbols
  - **Example**: `(age > 40) || (term < 10)`
    - Compound condition is true if `age > 40` or if `term < 10` or if both conditions are true
    - Note that the parenthesis could have been omitted in this expression this is true **because relational operators have higher precedence than logical operators**

- Notice with an || logical operator if Expression 1 is true the result of the || will always be true. Therefore the compiler does not need to check Expression 2 if Expression 1 is true. This is called **short circuit evaluation**

Expression 1	Expression 2	Expression 1 || Expression 2
false	false	false (0)
false	true	true (1)
true	false	true (1)
true	true	true (1)

Logical Operators

- **`NOT` Operator, `!`:**
  - Changes an expression to its opposite state
    - The ! operator reverses the "truth" of an expression. It makes a true expression false and a false expression true
  - If **`expressionA`** is true, then **`!expressionA`** is false
  - Example: !(1 > 0) evaluates to **`false`**

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

Logical Operators

TABLE 4.2 *Precedence of Relational and Logical Operators*

Operator	Associativity
`! unary - ++ --`	right to left
`* / %`	left to right
`+ -`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&&`	left to right
`||`	left to right
`= += -= *= /=`	right to left

Example: **(i == j) || (a < b) || complete**

Evaluate if: a = 12, b = 2, i = 15, j = 30, complete = 0

Evaluate if: a = 12, b = 2, i = 15, j = 30, complete = 5

Are the parenthesis needed? No because relational ops have higher precedence than logical operators (except for not (**!**)) (Let's demonstrate this is true in a program)

Logical Operators

- **Evaluate (i == j) || (a < b) || complete if: a = 12, b = 2, i = 15, j = 30, complete = 0**
- Are the parenthesis needed? No because relational ops have higher precedence than logical operators (Let's demonstrate this is true in a program)

- **Example: (Logical1.cpp – Example 5)**

```
#include <iostream>
using namespace std;

int main()
{
 //declare and initialize variables
 int a = 12;
 int b = 2;
 int i = 15;
 int j = 30;
 int complete = 0;

 cout << "(i == j) || (a < b) || complete is "
      << ((i == j) || (a < b) || complete) << endl;

  cout << "\ni == j || a < b || complete is "
      << (i == j || a < b || complete) << endl;

  system("PAUSE");
  return 0;
}
```

Logical Operators

- **Evaluate (i == j) || (a < b) || complete if: a = 12, b = 2, i = 15, j = 30, complete = 5**
- Why does it now evaluate to 1 instead of 0? (because any bool that is not 0 is true)

- **Example: (Logical1b.cpp – Example 6)**

```
#include <iostream>
using namespace std;

int main()
{
 //declare and initialize variables
 int a = 12;
 int b = 2;
 int i = 15;
 int j = 30;
 int complete = 5;

 cout << "(i == j) || (a < b) || complete is "
      << ((i == j) || (a < b) || complete) << endl;

  cout << "\ni == j || a < b || complete is "
      << (i == j || a < b || complete) << endl;

  system("PAUSE");
  return 0;
}
```

A Numerical Accuracy Problem

- Avoid testing equality of single and double-precision values (floating point numbers) and floating point variables using **==** operator (doubles, float, long double)
  - Tests fail because many decimals cannot be represented as accurately in binary
    - Many decimal numbers, such as 0.1, cannot be represented exactly in binary using a finite number of bits

- **For `real` operands:**
  - The expression

    ```
    operand_1 == operand_2
    ```

    should be replaced by

    ```
    abs(operand_1 - operand_2) < EPSILON
    ```

  - If this expression is true for very small **`EPSILON (such as .0000001),`** then the two operands are considered equal

  - abs is the absolute value math function (see Chapter 3)
    - (**`abs(x)`** returns the absolute value of x (in **`<cmath>`**))

  - Remember to use the abs math function you must include the file in your program

    ```
    #include <cmath>
    ```

The `if-else` Statement

- Selects a sequence of one or more instructions based on the results of *relational expression*

Review: Usually you will want to choose between blocks of code statements and will need curly braces

- **General form**:

```
if (expression) <- no semicolon here
   statement1;
else                          <- no semicolon here
   statement2;
```

  - If the value of expression is true, `statement1` is executed
  - If the value is false, `statement2` is executed

  - The relational expression is evaluated first.
    - ***If the value of the expression is nonzero (anything non zero is true in C++), statement1 is executed. If the value is zero, the statement after the keyword else is executed.***
    - Thus, ***one of the two statements (either statement1 or statement2) is always executed***, depending on the value of the expression. Note that the tested expression must be put in parentheses and that a semicolon is placed after each statement.

Review: you cannot have an else without an if, what happens if I want two statements to happen if the expression is true and I don't use curly braces?

The `if-else` Statement - Example: (TaxableIncome.cpp – Example 7)

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
 //declare named constants
 const double TAX_RATE_1 = .02;
 const double TAX_RATE_2 = .025;
 const double TAX_BRACKET_1 = 20000;

 //declare and initialize variables
 double taxable, taxes;

 //prompt for and get user input
 cout << "Please enter your taxable income:";
 cin >> taxable;

 //if taxable income is less than or equal to 20,000 the tax rate is .02
 if(taxable <= TAX_BRACKET_1)
 {
   taxes = taxable * TAX_RATE_1;
 }

 //otherwise calculate taxes with this formula for income of greater than 20,000
 else
 {
     taxes = TAX_RATE_2 * (taxable - TAX_BRACKET_1) + 400.00;
 }

 cout << fixed << setprecision(2) << "The taxes on " << taxable << " of taxable income  are $" <<
    taxes << endl;

 system("PAUSE");
 return 0;
}
```

Tip: always use curly braces with your statements for your if/else conditions. This makes your code more professional looking and readable and helps you never forget to use the curly braces (which are required if you have more than one statement.

The `if-else` Statement

- Program TaxableIncome.cpp run twice with different input data
  - Result 1:

```
Please type in the taxable income: 10000
Taxes are $ 200.00
```

  - Result 2:

```
Please type in the taxable income:  30000
Taxes are $ 650.00
```

The `if-else` Statement

- The if-else statement can be used to do basic data input checking in your program
  - **Example:** Program to prevent division by zero **(Div0.cpp – Example 8)**

```
#include <iostream>
using namespace std;

int main()
{
 float num1, num2, quotient;
 cout << "Enter a number: ";
 cin >> num1;
 cout << "Enter a second number: ";
 cin >> num2;

 if(num2 == 0)
 {
  cout << "Division by zero is not possible. \n";
  cout << "Please run the program again and enter ";
  cout << "\na number other than zero for the second number\n";
 }

 else
 {
  quotient = num1/num2;
  cout << "The quotient of  " << num1 << " divided by "
       << num2 << " is " << quotient << "." << endl;
 }

 system("PAUSE");
 return 0;
}
```

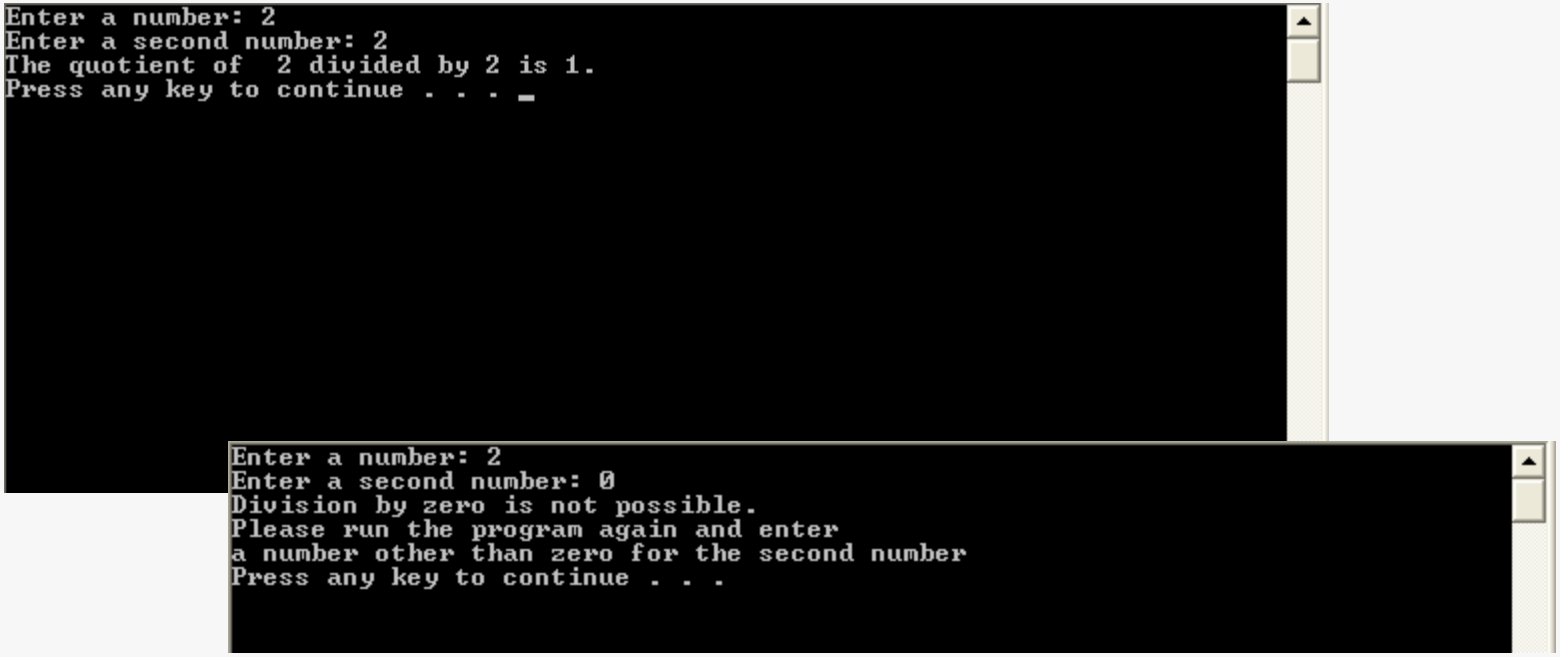
In this example your code would not work correctly without the curly braces

Let's experiment with what happens if the curly braces are forgotten

You can not have an else without an if – the else finds its if by looking for the ending curly braces of an if right before the else or an if above the statement right above the else

The `if-else` Statement

- The if-else statement can be used to do basic data input checking in your program
  - **Example:** Program to prevent division by zero
  - Program Results



The `if-else` Statement Example: (HighScore.cpp–Example 9)

- The if-else statement can be used with **boolean flags**
- A flag is a bool (boolean type) variable that signals when a condition exists by being true (any value other than 0) and signals when a condition doesn't exist by being false (0)

```
#include <iostream>
using namespace std;

int main()
{
    const int TOP_SCORE = 101999;

    int score;
    bool highScore = false;

    cout << "Please enter your score in the game:";
    cin >> score;

    if(score > TOP_SCORE)
    {
       highScore = true;
    }

    //notice this is the same as writing if(highScore == true)
    if(highScore)
    {
       cout << "CONGRATULATIONS YOU HAVE THE NEW TOP SCORE!" << endl;
    }

    else
    {
       cout << "Please try again." << endl;
    }

  system("PAUSE");
  return 0;
}
```

Note: You CAN have an if without an else but you can NOT have an else without an if

Compound Statements

- In the case that there is only one statement after the if/else no curly braces HAVE to be used (they can AND SHOULD be used though) around the statements that are to be executed after the if and else
  - Look at HighScore.cpp – the curly braces weren't required
- In the case that there is more than one statement after the if/else curly braces are REQUIRED after the if and else conditions in order to tell the compiler which statements go with the if and the else
- IF you forget the curly braces, the compiler will only associate the first program statement after the if or else with that if/else
  - Rerun Div0.cpp without the curly braces after the else
- The area enclosed by these curly braces is known as a **statement block**.
  - **If the if statement is true** a block containing one or more statements is executed
  - **If the if statement is false** another group of statements (the block of statements following the else statement in this case) is executed
  - While the curly braces marking the block of code to be executed is not required if you have only one statement after the if or else, I generally tend to use the curly braces no matter what to make the code easier to follow.
    - It can also be helpful in debugging
    - Every { must have a }

Compound Statements

FIGURE 4.2 *A Compound Statement Consists of Individual Statements Enclosed Within Braces*

```
if (expression)
{
  statement1;    // as many statements as necessary
  statement2;    // can be put within the braces
  statement3;    // each statement must end with a ;
}
else
{
  statement4;
  statement5;
       .
       .
       .
  statementn;
}
```

Block Scope

- **Block of Code**: All statements contained within a compound statement

- ***Any variable declared within a block has meaning only between its declaration and the closing braces of the block***
  - This means code inside the inner braces can look out but code in the outer braces cannot look in
    - code always looks for a variable inside its own block (set of curly braces first) and then if it can't find the variable starts looking outward block by block
    - ***IMPORTANT:*** The GENERAL rule is NOT to limit the scope of a variable by declaring it in a block unless you have a REASON too – MOST variables (almost all in our class) should be declared at the top of main (or of the function you are writing when we write more than main) in order to be visible by ALL code in the function
      - This example is for illustration it is not good code

- Example with two blocks of code

Block Scope (continued) – (block.cpp – Example 10)

```
#include <iostream>
using namespace std;

int main()
{
    { // start of outer block - RANDOM CURLY BRACES LIKE THIS ARE BAD CODE!!!!
      int a = 25;
      int b = 17;
      cout << "The value of a is " << a << " and b is " << b << endl;

      { // start of inner block
       double a = 46.25;
       int c = 10;
       cout << "\na is now " << a
            << " b is now " << b
            << " and c is " << c << endl;
      } // end of inner block

      cout << "\na is now " << a << " and b is " << b << endl << endl;

    } // end of outer block

   return 0;
}
```

Block Scope (continued)

- **Output of Block Scope example**:

```
The value of a is 25 and b is 17
a is now 46.25 b is now 17 and c is 10
a is now 25 and b is 17
```

- The first block of code defines two variables a and b which may be used anywhere within the block including within any block contained within the block

- The second block (the inner block) declares two new variables a and c
  - Because no b was declared in the inner block the b from the outer block prints in the second `cout` statement
    - If no variable is defined in a block where it is used the compiler attempts to find the variable in the immediate outer block and goes outward block by block until it finds it or reaches the outermost block and returns an invalid result

- Finally when a and b are displayed the third time in the outer block the outer block's a and b are used. The outer block cannot see the variables in the inner block. If an attempt were made to access c anywhere in the outer block, the compiler would issue an error message saying there was an undefined symbol.

Block Scope (continued)

- **Common programming practice**: place opening brace of a compound statement on the same line as **`if`** and **`else`** statements

```
if (tempType == 'f') {
  celcius = (5.0 / 9.0) * (temp - 32);
  cout << "\nThe equivalent Celsius temperature is"
          << celsius << endl;
}
```

Block Scope (continued)

- **The traditional format:**

```
if (tempType == 'f')
{
   celsius = (5.0 / 9.0) * (temp - 32);
   cout << "\nThe equivalent Celsius temperature is "
        << celsius << endl;
}
```

One-Way Selection

- A modification of **`if-else`** that omits *else* part
  - **`if`** statement takes the form:

    `if (`*`expression`*`)`

      ***`statement`*`;`**
- Modified form called a one-way statement
  - The statement following **`if (`*`expression`*`)`** is executed only if the expression is true
  - The statement may be a compound statement

One-Way Selection

- **Example: (CarMileage.cpp – Example 11)**

```
#include <iostream>
using namespace std;

/*This program tests whether a car is over the mileage limit to be covered
by a warranty*/
int main()
{
   const double LIMIT = 3000.0;

   int idNum;
   double miles;

   cout << "Please type in the car number followed by a space followed by the mileage:";
   cin >> idNum >> miles;

   //one way if
   if(miles > LIMIT)
   {
      cout << "Car Number " << idNum << " is over the mileage limit." << endl;
   }

   cout << "End of program output." << endl;

   system("PAUSE");
   return 0;
}
```

When you plan your program you're going to need to choose – the proper logical structure. You should not choose if if if when logic says if-else makes more sense.

Review: how do you decide between these two structures? What is the logic?

One-Way Selection (continued)

- Program **CarMileage.cpp** run twice with different input data
  - Result 1:

```
Please type in car number and mileage: 256  3562.8
   Car 256 is over the limit.
End of program output.
```

  - Result 2:

```
Please type in car number and mileage: 23  2562.8
End of program output.
```

Problems Associated with the `if-else` Statement

- **Most common problems**:
  - Misunderstanding what a relational expression is
    - A relational expression evaluates to 1 or 0 depending if it is true or false
    - An expression that evaluates to 0 evaluates to false (0) and ***an expression that evaluates to any positive number evaluates to true (1)***
  - **Using the assignment operator, =, in place of the relational operator, ==**

- **Example**:
  - Imagine you initialized the age variable as follows: `age = 18`
  - Later in your program you have an if statement that starts like this:

```
if(age = 30)
{
   program statement(s);
}
```

  - The expression `(age = 30)` sets `age` to `30`
    - Does not compare `age` to 30
    - Has a value of 30 which is a **true** result (in C++ anything not 0 is true)
      - This means the statements following the if statement would execute
    - Produces invalid results if used in `if-else` statement
      - Notice it still compiles, it just produces a result other than what you might want

Problems Associated with the `if-else` Statement

- Example continued:
  - This is the use of the correct relational expression: `age == 30`

```
if(age == 30)
{
  program statement(s);
}
```

  - The expression in this if statement compares `age` to `30`
    - has a value of 0 (false)
  - This expression will produce a valid test in an `if-else` statement in the example where age was initially initialized to 18

Problems Associated with the `if-else` Statement

Example: (Birthday.cpp – Example 12)

--Two unintended things happened (an unintended assignment and an incorrect relational result)– BE CAREFUL with == and =

```
#include <iostream>
using namespace std;

int main()
{
    int age = 40;

   //this not only causes the wrong if statement to
   //execute but assigns the value 30 to age
   if(age = 30)
   {
        cout << "Happy Birthday Incorrect." << endl;
        cout << "Age is " << age << endl;
   }

   if(age == 40)
   {
        cout << "Happy Birthday Correct." << endl;
   }

   system("PAUSE");
   return 0;
}
```

PROGRAM OUTPUT:

```
Happy Birthday Incorrect.
Age is 30
Press any key to continue . . .
```

Nested selection Statements – Example (GradDisc.cpp– Example 13)

- **Concept:** A nested if statement is an if statement in the conditionally executed code of another if statement

```
#include <iostream>
using namespace std;

int main()
{
  char employed, recentGrad;

  cout << "Answer the following questions\n";
  cout << "with either Y for yes or N for no.\n\n";
  cout << "Are you employed? ";
  cin >> employed;

  if(employed == 'Y')
  {
     cout << "Have you graduated from college in the past 2 years? ";
     cin >> recentGrad;
     if(recentGrad == 'Y')
     {
        cout << "You qualify for the special interest rate. \n";
     }
     else
     {
       cout << "You must have graduated in the last two years to qualify.\n";
     }
  }
  else
  {
      cout << "You must be employed to qualify\n";
  }
  system("PAUSE");
  return 0;
}
```

HW #2 should use nested selection statements

Nested `if` Statements

- Example Results

```
Answer the following questions
with either Y for yes or N for no.

Are you employed? Y
Have you graduated from college in the past 2 years? Y
You qualify for the special interest rate.
Press any key to continue . . .
```

```
Answer the following questions
with either Y for yes or N for no.

Are you employed? Y
Have you graduated from college in the past 2 years? N
You must have graduated in the last two years to qualify.
Press any key to continue . . .
```

```
Answer the following questions
with either Y for yes or N for no.

Are you employed? N
Have you graduated from college in the past 2 years? Y
You must be employed to qualify
Press any key to continue . . .
```

Nested `if` Statements

- When debugging a program with nested **`if/else`** statements it is important to know which if statement each else belongs to
  - The rule is that **an `else` goes with the last `if` statement that doesn't have it's own `else`**
  - This is easiest to see when all if statements are properly indented and each else is lined up with the if it belongs to
    - See how the example was formatted
  - Nested if statements can become very long and complex so these visual clues are important

The if/else if statement (else-if chain)

- **Concept:** The if/else if statement is a chain of if statements. They perform their test one after another until one of them becomes true
- **Format:**

```
if (expression_1)
      statement1;
else if (expression_2)
      statement2;
else
      statement3;
```

- The else here is like a default in that only if none of the ifs or else ifs are true does it execute
  - The else is not required
  - It is called a **trailing case**
  - If the **else-if chain** doesn't have a trailing case it is possible none of the statements in the chain get executed
- **How it works:**
  - The expression after the **`if`** is evaluated,
    - if it is true its statements execute and the if-else chain terminates
    - if it is false the expression after the **`else if`** immediately following it is evaluated
      - if it is true its statements execute and the if-else chain terminates
      - if it is false the expression after the **`else if`** immediately following it is evaluated
        - This pattern continues until an else if expression evaluates to true or the compiler reaches the end of the if-else chain
- Chain can be extended indefinitely by making last statement another **`if-else`** statement

The `if-else` Chain – Example (MarCodeMenu.cpp – Example 14)

```
#include <iostream>
using namespace std;

int main()
{
  char selection;

  cout << "Marital Code Menu" << endl;
  cout << "m) Married" << endl;
  cout << "s) Single" << endl;
  cout << "w) Widowed" << endl;
  cout << "d) Divorced" << endl;
  cout << "Please make a selection: ";

  cin >> selection;

  if(selection == 'm')
     cout << "You selected Married." << endl;

  else if(selection == 's')
     cout << "You selected Single." << endl;

  else if(selection == 'w')
     cout << "You selected Widowed." << endl;

  else if(selection == 'd')
     cout << "You selected Divorced." << endl;

  else
     cout << "You made an Invalid Choice." << endl;

  system("PAUSE");
  return 0;
}
```

The only reason the curly braces aren't here is because I have limited space on the slides, this is not a very readable or well structured program as is, I should add curly braces.

The `switch` Statement

- **Concept:** The switch statement lets the value of a variable or expression determine where the program will branch to
  - The switch statement provides an **alternative to the if-else chain for cases that compare the value of an integer expression to a specific value**.

- **Format:**

```
switch (expression)
{                   // start of compound statement
   case value_1: <- terminated with a colon
     statement1;
     statement2;
     break;
   case value_2:  <- terminated with a colon
     statement;
     break;
   default:                <- terminated with a colon
     statement;
}                   // end of switch and compound
                    // statement
```

The `switch` Statement

- **Four new keywords used**:
  - `switch`, `case`, `default` and `break`
- **Function:**
  - Expression following `switch` is evaluated
    - Must evaluate to an **integer result** (**remember char evaluates to an integer**)
  - Result compared sequentially to alternative `case` values until a match found
  - If a match is found, statements following matched `case` are executed
  - When `break` statement reached, `switch` terminates
  - Any number of case labels may be contained within a switch statement, in any order. If the value of the expression does not match any of the case values, however, no statement is executed unless the keyword `default` is encountered. The keyword default is optional and operates the same as the last else in an if-else chain.
    - If no match found, default statement block is executed
- Once an entry point has been located by the switch statement, all further case evaluations are ignored, and execution continues through the end of the compound statement unless a `break` statement is encountered.
  - This is the reason for the `break` statement, which identifies the end of a particular case and causes an immediate exit from the switch statement. Thus, just as the word case identifies possible starting points in the compound statement, the break statement determines terminating points. **If the break statements are omitted, all cases following the matching case value, including the default case, are executed.**

IMPORTANT: `break` ***should ONLY be used in a*** `switch` ***statement***, using it in other structures such as if-else, if, or if-else if structures and loops is spaghetti code (unreadable, not maintainable, hard to re-design), you should use the logic of the structure to exit NEVER use a `break`, ***using*** `break` ***elsewhere in code WILL result in a loss of points***

The `switch` Statement – Ex. (MathCalc.cpp – Example 15)

```
#include <iostream>
using namespace std;
int main()
{
      int opselect;
      double fnum, snum;

      cout << "Please type in two numbers: ";
      cin >> fnum >> snum;

      cout << "\n1)Addition";
      cout << "\n2)Multiplication";
      cout << "\n3)Division";
      cout << "\nPlease make a selection:  ";
      cin >> opselect;

      switch(opselect)
      {
         case 1:
              cout << "The sum of the numbers is " << fnum + snum << endl;
              break;

         case 2:
              cout << "The multiplication of the numbers is " << fnum * snum << endl;
              break;

         case 3:
              if(snum != 0)
                 cout << "The division of the numbers is " << fnum/snum << endl;

              else
                 cout << "Division by zero is not allowed." << endl;
              break;
      }
     system("PAUSE");
     return 0;
}
```

The `switch` Statement – Ex. (MathCalc2.cpp – Example 16)

```
#include <iostream>
using namespace std;
int main()
{
      char opselect;
      double fnum, snum;

      cout << "Please type in two numbers: ";
      cin >> fnum >> snum;

      cout << "\na)Addition";
      cout << "\nb)Multiplication";
      cout << "\nc)Division";
      cout << "\nPlease make a selection:  ";
      cin >> opselect;

      switch(opselect)
      {
         case 'a':
              cout << "The sum of the numbers is " << fnum + snum << endl;
              break;

         case 'b':
              cout << "The multiplication of the numbers is " << fnum * snum << endl;
              break;

         case 'c':
              if(snum != 0)
                 cout << "The division of the numbers is " << fnum/snum << endl;

              else
                 cout << "Division by zero is not allowed." << endl;
              break;
      }
      system("PAUSE");
      return 0;
}
```

The `switch` Statement – Ex. (MathCalc3.cpp – Example 17)

```
#include <iostream>
using namespace std;
int main()
{
      char opselect;
      double fnum, snum;

      cout << "Please type in two numbers: ";
      cin >> fnum >> snum;

      cout << "\na)Addition";
      cout << "\nb)Multiplication";
      cout << "\nc)Division";
      cout << "\nPlease make a selection:  ";
      cin >> opselect;

      //ALL OF THE BREAK STATEMENTS ARE GONE SO IT FALLS THROUGH THEM ALL ONCE
      //ONCE CASE MATCH IS FOUND
      switch(opselect)
      {
         case 'a':
              cout << "The sum of the numbers is " << fnum + snum << endl;

         case 'b':
              cout << "The multiplication of the numbers is " << fnum * snum << endl;

         case 'c':
              if(snum != 0)
                 cout << "The division of the numbers is " << fnum/snum << endl;

              else
                 cout << "Division by zero is not allowed." << endl;
      }
      system("PAUSE");
      return 0;
}
```

Ready for HW #2

- The rest of this lecture is on randoms which you don't need for HW #2 (you will need them for HW #3) so we are going to do the following and then come back to randoms
  - **Intro to String Lecture**
  - **Tips on HW #2**

`rand` – Random Numbers in C++

- Used in Problems requiring statistical models
  - Business Programs simulating patters such as telephone usage and traffic flow
  - Various Gaming Scenarios

- These type of models require the use of ***random numbers*****:** numbers whose order can't be predicted

- In practice there are truly no random numbers
  - Dice are never perfect, cards are never shuffled perfectly
  - Digital computers can only handle numbers in a specified range
  - The best that can be done is the generation of ***pseudorandom numbers*****:** these numbers are sufficiently random for the task at hand

`rand` – Random Numbers in C++

- All C++ compilers provide two functions for creating random numbers:
- **`rand()`:** This function returns a pseudo-random integral number (an **`int`**) in the range 0 to **`RAND_MAX (not including RAND_MAX)`** .
  - This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a starting value (the starting value is called a ***seed***) to generate the series, which should be initialized to some distinctive value using the function **`srand`**
  - **`RAND_MAX`** **is a constant defined in `<cstdlib>` .** Its default value may vary between implementations (meaning compilers) but it is granted to be at least 32767.
- **`srand(unsigned int)`** **:** This function accepts an unsigned **`int`** as an argument and uses it to set the seed (starting value) for the rand function.
  - If **`srand`** is not set each the rand function will produce the same **sequence** of numbers each time because the same starting number is being used in the algorithm generating the random sequence
  - The most common way to use **`srand`** is as follows:
    - **`srand ((unsigned int)time(NULL));`**
    - Here the argument to **`srand`** is a call to the time function with a NULL argument.
      - The time function reads the computer's internal computer in seconds
      - The **`srand`** function uses this time converted to an unsigned **`int`** to **`initialze`** the seed (starting value of the random number generator for rand()
- You must **`#include <cstdlib>`** when using the random number functions
- To use system time you must also **`#include <ctime>`**

Caution: you should only set the seed of the random algorithm ONCE at the beginning of the program you will use randoms in

`rand` – Random Numbers in C++

- Most of the time you will want to generate a random number in a specified range
- You can do this using the `%` operator

  - `number = rand() % 100;  //number  is in the range 0 to 99`
    - The `mod %` operator returns the remainder of integer division. When the integer returned by `rand()` is divided by 100 the remainder will be a value between 0 and 99

  - `number =  rand() % 100 + 1// is in the range 1 to 100`
    - Since the remainder of an integer divided by 100 is between 0 and 99 if we want the range of our random number to be between 1 and 100 we just need to add one to the remainder

rand – Random Numbers in C++ - Ex. (Guess.cpp – Example 18)

```
#include <cstdlib>
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
   int secretNum;
   int guess;
   bool hasWon = false;

   /* initialize random seed: */
   srand ((unsigned int)time(NULL));

  /* generate secret number: */
  secretNum = rand() % 25 + 1;

  cout << "You will get three guesses." << endl;

  cout << ("Guess the number (1 to 25): ");
  cin >> guess;

  if (secretNum <  guess)
     cout << "The secret number is lower" << endl;

  else if (secretNum > guess)
     cout << "The secret number is higher" << endl;

  else
    hasWon = true;
```

- **CONTINUED ON NEXT PAGE**

rand – Random Numbers in C++ - Ex. (Guess.cpp – Example 18)

- CONTINUED FROM PREVIOUS PAGE

```
//only have them guess a second time if they didn't win
  if(!hasWon)
  {
     cout << ("Guess the number (1 to 25): ");
     cin >> guess;

     if (secretNum <  guess)
        cout << "The secret number is lower" << endl;

     else if (secretNum > guess)
        cout << "The secret number is higher" << endl;

     else
        hasWon = true;
  }

   //only have them guess a third time if they didn't win
  if(!hasWon)
  {
     cout << ("Guess the number (1 to 25): ");
     cin >> guess;

     if (secretNum <  guess)
        cout << "The secret number is lower" << endl;

     else if (secretNum > guess)
        cout << "The secret number is higher" << endl;

     else
        hasWon = true;
  }
```

rand – Random Numbers in C++ - Ex. (Guess.cpp – Example 18

- CONTINUED FROM PREVIOUS PAGE

```
  if(hasWon)
    cout << "Congratulations!  The secret number was " << secretNum << endl;

  else
     cout << "Sorry, the secret number was " << secretNum << endl;

  system("PAUSE");
  return 0;
}
```

Common Programming Errors

- Using the assignment operator , `=`, in place of the relational operator, `==`
- Assuming that the `if-else` statement is selecting an incorrect choice when the problem is really the values being tested
- Using nested `if` statements without including braces to clearly indicate the desired structure

Summary

- Relational Expressions (conditions):
  - Are used to compare operands
  - A condition that is true has a value of 1
  - A condition that is false has a value of 0
- More complex conditions can be constructed from relational expressions using C++'s logical operators, `&&` (AND), `||` (OR), and `!` (NOT)
- `if-else` statements select between two alternative statements based on the value of an expression

Summary (continued)

- `if-else` statements can contain other `if-else` statements
  - If braces are not used, each `else` statement is associated with the closest unpaired `if`
- **Compound Statement:** any number of individual statements enclosed within braces
- Variables have meaning only within the block where they are declared
  - Includes any inner blocks
- **`switch` Statement:** multiway selection statement
  - The value of an integer expression is compared to a sequence of integer or character constants or constant expressions
  - Program execution transferred to first matching case
  - Execution continues until optional break statement is encountered