Introduction

This lab will provide students with an introduction to the fundamentals of conditional selection in the C++ programming language. The focus of this lab is on how to solve simple problems using the C++ “if” statement and relational operators. At the end of this module, students will…

- Be able to solve simple problems using C++ “if” and “if-else” statements
- Understand the structure of simple C++ “if” and “if-else” statements
- Understand and be able to apply simple conditional expressions
- Understand and apply the basic concepts behind C++ relational operators

These exercises have two different types of answers. Some questions ask for answers to specific questions; you should collect these answers in a Microsoft Word document. Other exercises ask you to write code; for these exercises, structure your storage drive in the following manner. There should be a palms folder, and within that folder there should be a LabI folder. All coding exercises (and related folders) for this lesson should be placed in the LabI folder. Let’s begin!

Part 1: Conditional Selection in C++

Sequential execution of statements is the standard flow of control in C++, or the way in which statements are executed. A program executes starting with the first statement in the main() function and terminates with the last statement in main(), if all goes well. Individual expressions are executed left-to-right.

A well-designed program needs to respond to different types of input. In order for the program to execute differently for different types of input, conditional language constructs are needed. These alter the flow-of-control based on the result of evaluating a Boolean (true/false) expression. The flow of control mechanism known as conditional selection permits alternate sections of code to be executed based on a conditional test. In this worksheet, we will learn how to use the C++ if statement to perform conditional selection.

If Statements in C++

A well-designed computer program needs to respond to different types of inputs and situations. For example, consider the case of an Automatic Teller Machine (ATM). Whether you realized it or not, ATM machines make decisions based on what the user inputs into the machine — i.e. the specific task requested. Suppose you wanted to take $40 out of your checking account. After requesting a cash withdrawal and specifying $40, the ATM must make a decision. If you have enough money in your checking account to permit the transaction, the ATM will dispense the
money; otherwise, the ATM will not dispense the money and inform you that you have insufficient funds for the withdrawal.

The decisions made by an ATM can be thought of as conditional selection. **Conditions** are expressions which evaluate to one of two possible values: **true** or **false**. You can think of conditional expressions like a fork in the road. Based on the expression’s value, only one of two possible pathways in a program will be “taken” or executed. At this point, you should watch and listen to the Flowcharts and Conditional Selection animation to reinforce these concepts.

The most common conditional selection construct in C++ is the "if" **statement**, which executes a statement or a series of statements only if a specified condition is true. Depending on the value of a specific condition – true or false – an if statement can choose among alternative sequences of code to execute, or even choose to execute no code at all. The general form is as follows:

```cpp
if ( /* condition */ )
{
    // one or more statements to execute if the
    // value of the condition is true
}
else
{
    // one or more statements to execute if the
    // value of the condition is false
}
```

The else clause is strictly optional. The following example shows a simple if statement:

```cpp
if ( account_balance >= amount_requested )
{
    // call the function dispenseMoney...
    dispenseMoney(40.00);
}
```

Adding an else clause to this example would produce two possible paths of execution:

```cpp
if ( account_balance >= amount_requested )
{
    // call the function dispenseMoney...
    dispenseMoney(40.00);
}
else
{
    // output a message to the user...
    cout << "You do not have sufficient funds!" << endl;
}
```
Notice how the `if` statement includes one or more statements within it. This represents a **compound statement**; even though the `if` statement represents a single statement, it can contain multiple statements within it which only execute if the conditional expression is true. If the conditional expression is not true, these nested statements will not execute.

Watch and listen to the [If-Else Statements C++ podcast below](https://example.com/):  

```cpp
// job classification
string classification = "Part Time";
if ( hours_worked > 37.50 )
{
   // the job is classified as full-time
   classification = "Full Time";
}

If Statements in C++
```

**Relational Operators**

Conditional selection constructs alter the flow-of-control based on the result of evaluating a conditional expression. To construct these expressions, all programming languages have built-in binary **relational operators** for building conditional expressions. C++ has six relational operators, all of which are derived from mathematics:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
</tbody>
</table>

These operators are **binary** in the sense that they require two operands (one on the left of the operator, one on the right). The operator compares these operands and produces a Boolean (true or false) result. For example, the conditional expression `(account < 2000.00)` is a Boolean
expression which is true only if the value of account is less than 2000. If account is not less than 2000, the expression is false. Conditional expressions should always be enclosed in parentheses to avoid any ambiguity.

A common error for beginning C++ programmers involves the use of the equality operator. Unlike what we are used to from mathematics, the equality operator is \(==\), not \(=\). This was done because the \(=\) is used for assignment, i.e. placing a value into a variable. Always be sure to use \(==\) to check for equality, because the compiler will not tell you of your mistake!

**Exercise 0:** Create a new C++ code file called EvenOdd.cpp and type the following code into the new file:

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

int main()
{
    int user_input = 0; // stores the user input
    cout << "Please enter an integer: "; // prompt for number
    cin >> user_input; // input number
    // echo-print the input...
    cout << endl << "You entered " << user_input << "." << endl;

    if ( (user_input % 2) == 0 )
    {
        // this is an even number...
        cout << "This is an even number." << endl;
    }
}
```
if ( (user_input % 2 ) != 0 )
{
    // this is an odd number...
    cout << "This is an odd number." << endl;
}
// return success to the operating system...
return 0;

Compile and run the program. Try different input data to verify it works as it should; alternate entering odd and even numbers.

Computing Payroll Amounts

“When you come to a fork in the road, take it.” – Yogi Berra

The amount you are paid for doing work is always more than you actually take home. Taxes and other deductions reduce the size of your gross pay (the total amount of money you make) to compute your net pay (the total amount you actually take home). Computing the net pay depends on many different variables which are different for each person. As a result, computer programs which produce payroll checks make heavy use of conditional selection.

Exercise 1: Create a new C++ Project called Payroll. Add a new C++ code file called Payroll.cpp to the project, and type the following code into the new file:

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

```

```
```
```cpp
// FUNCTION NAME: main
// PARAMETERS: None
// RETURN TYPE: int
// PURPOSE: Entry point for the application.

int main()
{
    string name; // stores the user's name
    double hourly_rate=0.0, gross_pay = 0.0, net_pay = 0.0; // stores results
    int hours = 0; // stores the number of hours

    cout << "Please enter your first name: "; // prompt for name
    cin >> name; // input name
    cout << "Please enter the number of hours worked: "; // prompt for hours
    cin >> hours; // input hours
    cout << "Please enter your hourly pay rate: "; // prompt for pay
    cin >> hourly_rate; // input pay

    cout.setf(ios::fixed); // set up numeric output for real numbers
    cout.precision(2); // set up numeric output for 2 digits of precision

    // echo-print the input...
    cout << endl 
        << "Welcome, " << name << ". You worked " << hours << " hours"
        << " for $" << hourly_rate << " per hour. " << endl;

    if (hours <= 40)
    {
        // compute the gross pay (no overtime)...
        gross_pay = hourly_rate * hours;
    }

    // output the results...
    cout << "Your gross pay is: "$ << gross_pay << endl; // output gross pay

    // return success to the operating system...
    return 0;
}
```

Compile and run the program. Try different sets of input data to verify it works as it should.

**Exercise 2:** Enhance the Payroll program by adding an else clause to the if statement. The code within the else clause must compute the gross pay for an overtime situation. When workers work overtime, they get paid time and a half for all hours over 40. The else clause should appear as follows:

```cpp
if (hours <= 40)
```
Compile and run the program. Try different sets of input data to verify it works as it should.

**Exercise 3:** Add a second `if` statement to the Payroll program which calculates the net pay and displays it to the user. The net pay is the gross pay if the gross pay is less than $500.00. If the gross pay is greater than $500.00, the net pay is the gross pay minus a 25% tax. Your `cout` statement for the net pay should be placed after the existing one for the gross pay). Compile and run the program. Try different sets of input data to verify it works as it should.

**Part 2: Making Decisions and Computing Results**

Computing mathematical results on a computer is dependent upon the values used in the calculation. Consider the quadratic equation, which should be familiar to you from middle school algebra class. If you recall, the general form of a quadratic equation in mathematics is:

\[ y = ax^2 + bx + c \]

Where \( a \), \( b \), and \( c \) are constant coefficients. Remember that a quadratic equation is a polynomial of the second degree – i.e. the value of the coefficient on the \( x^2 \) term is non-zero. If we wanted to calculate values dealing with quadratic equations on the computer, we could write a program to input the three coefficients \((a, b, \text{ and } c)\) and then make sure that the input values represent a quadratic equation by checking to see if \( a > 0 \). If the user entered a zero for the first coefficient, our program would need to alert the user of his or her mistake.

**Exercise 4:** Create a new C++ Project called Quadratic. Add a new C++ code file called `Quadratic.cpp` to the project, and type the following code into the new file:
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
    double a = 0.0, b = 0.0, c = 0.0;  // stores the coefficients

    // prompt the user to enter three coefficients, separated by spaces...
    cout << "Please enter the coefficients, separated by spaces: ";
    cin >> a >> b >> c;  // input the three values

    // echo-print the input...
    cout << endl
        << "You entered a = " << a << ", b = " << b
        << ", c = " << c << "." << endl;

    if ( a == 0 )
    {
        // not a quadratic equation, display an error message...
        cout << "ERROR: the first coefficient must not be zero." << endl;
    }
    else
    {
        // a valid quadratic equation, display an error message...
        cout << "The coefficients are valid." << endl;
    }

    // exit the program with success (0 == success)
    return 0;
}
Compile and run the program twice, using the following values for inputs: \( a=1.0, b=3.0, c=4.0 \) on the first run, \( a=0.0, b=3.4, c=35.7 \) on the second. What happens when you run the program?

Exercise 5: Enhance the Quadratic program in the following manner. If the user enters a valid set of coefficients (the else” case) compute the value of the discriminant \( d \), defined as:

\[
d = b^2 - 4ac
\]

If the value of the discriminant is negative, inform the user that the roots are complex. If the value is non-negative, inform the user the roots are not complex. Compile and run the program multiple times. Verify the program acts as it should.

Exercise 6: Enhance the Quadratic program in the following manner. If the roots are not complex, compute the two roots of the quadratic equation using the formula:

\[
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

Once the two roots are calculated, display them to the screen with a descriptive message. At this point you should go back and change the PURPOSE comment at the top of the file – you have significantly changed what this program does!

Here are a few test cases that may be helpful. To have the results show as four digits of precision, place the following two lines of code immediately before your first “if” statement:

```cpp
cout.setf(ios::fixed); // set up numeric output for real numbers
cout.precision(4);    // set up numeric output for 4 digits of precision
```

Test Case 1: \( a= 1.0, b = 4.0, c = 3.0 \)

Please enter the coefficients, separated by spaces: 1.0 4.0 3.0
You entered a = 1.000, b = 4.000, c = 3.000.
The coefficients are valid.
The roots are not complex.
The roots of the quadratic equation are -1.000 and -3.000.
Test Case 2: \( a = 4.0, b = 3.5, c = 1.7 \)

Please enter the coefficients, separated by spaces: 4.0 3.5 1.7
You entered \( a = 4.000, b = 3.500, c = 1.700 \).
The coefficients are valid.
The roots are complex numbers.

Test Case 3: \( a = 0.0, b = 4.1, c = 56.8 \)

Please enter the coefficients, separated by spaces: 0.0 4.1 56.8
You entered \( a = 0.000, b = 4.100, c = 56.800 \).
ERROR: the first coefficient must not be zero.

Test Case 4: \( a = 1.0, b = 3.2, c = 0.0 \)

Please enter the coefficients, separated by spaces: 1.0 3.2 0.0
You entered \( a = 1.000, b = 3.200, c = 0.000 \).
The coefficients are valid.
The roots are not complex.
The roots of the quadratic equation are 0.000 and -3.200.

Part 3: Diagnosing Blood Pressure Problems

While each of us has had a reading of our blood pressure taken, many of us may not know what blood pressure actually means. *Blood pressure* is expressed as a ratio of the maximum (*systolic*) and minimum (*diastolic*) pressure found in the aorta during one cardiac cycle. These pressure values are measured in millimeters of mercury (mmHg). The following link may prove useful in visualizing these concepts:

Cardiac Cycle (University of Utah Medical School)
Animation showing the cardiac cycle (Requires Adobe Shockwave).

Determining abnormal blood pressure is critical for diagnosing and averting a number of serious health problems (e.g. heart disease and stroke). Doctors therefore make decisions about the potential for serious health problems based on a few simple calculations. Two simple calculations of interest are *pulse pressure* (PP) and *mean arterial pressure* (MAP).

The *pulse pressure* of an individual can be computed as the difference between the systolic and diastolic pressures. The *mean arterial pressure* is the average pressure during the cardiac cycle. The MAP of a normal, resting heart can be computed by the formula:

\[
MAP = DP + \frac{1}{3} PP
\]
where DP is the diastolic pressure and PP is the pulse pressure.

**Exercise 7:** Create a new C++ Project called `ArterialPressure`. Write a C++ program which reads the systolic and diastolic pressures (both integers) from the user and displays the corresponding pulse pressure. Assume all values are in mmHg. If the pulse pressure is greater than 80 mmHg, inform the user that their pulse pressure is high. Use a simple `if` statement (no `else` clause). **Create multiple functions besides main – two for input, one for computation, and one for output.**

**Exercise 8:** Enhance the `ArterialPressure` program to compute and display the mean arterial pressure (a real-number value). If the mean arterial pressure is below 60 mmHg, alert the user that he or she should seek medical assistance; else, inform the user that their mean arterial pressure is within acceptable limits.

Here are a few test cases that may be helpful.

**Test Case 1: Normal Blood Pressure (120/80)**

<table>
<thead>
<tr>
<th>Enter your systolic pressure: 120</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter your diastolic pressure: 80</td>
</tr>
<tr>
<td>Your pulse pressure is: 40.00</td>
</tr>
<tr>
<td>Your mean arterial pressure is: 93.33</td>
</tr>
<tr>
<td>Your mean arterial pressure is within acceptable limits.</td>
</tr>
</tbody>
</table>

**Test Case 2: High Blood Pressure (170/110)**

<table>
<thead>
<tr>
<th>Enter your systolic pressure: 170</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter your diastolic pressure: 110</td>
</tr>
<tr>
<td>Your pulse pressure is: 60.00</td>
</tr>
<tr>
<td>Your mean arterial pressure is: 130.00</td>
</tr>
<tr>
<td>Your mean arterial pressure is within acceptable limits.</td>
</tr>
</tbody>
</table>

**Test Case 3: High Pulse Pressure (170/80)**

<table>
<thead>
<tr>
<th>Enter your systolic pressure: 170</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter your diastolic pressure: 80</td>
</tr>
<tr>
<td>Your pulse pressure is: 90.00</td>
</tr>
<tr>
<td>Your pulse pressure is high.</td>
</tr>
<tr>
<td>Your mean arterial pressure is: 110.00</td>
</tr>
<tr>
<td>Your mean arterial pressure is within acceptable limits.</td>
</tr>
</tbody>
</table>
Test Case 4: Low Mean Arterial Pressure (80/40)

Enter your systolic pressure: 80
Enter your diastolic pressure: 40
Your pulse pressure is: 40.00
Your mean arterial pressure is: 53.33
You should seek medical assistance.

Part 4: Vocabulary Words

You will be assessed on your knowledge of the following terms and concepts as well as the C++ concepts discussed in this document. As a result, you are encouraged to review these terms in the preceding document.

**Boolean** – true or false.

**Conditions** – expressions which evaluate to one of two possible values: true or false.

**Conditional Selection** – a C++ language construct that permits alternate sections of code to be executed based on a conditional (true/false) test.

**Compound Statement** – a C++ statement that contains multiple statements within it.

**Operand** – a C++ variable, expression, or literal that can be used with a C++ operator. For example, in the expression 3+5, the numbers 3 and 5 are operands and the + symbol is the operator.

**Operator** – a C++ symbol which allows a specific operation to be performed on one or more operands.

**Relational Operators** – binary operators which test the relationship between two values. In C++, the operators are ==, !, >, <, >=, and <=. These operators produce a Boolean result when applied to two operands.

**Sequential execution** – the standard flow of control in C++, or the way in which statements are executed. A program executes starting with the first statement in the `main()` function and terminates with the last statement in `main()`, if all goes well. Individual expressions are executed left-to-right.