# Lecture 7 — Selection Statements

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 27, 2016

Acknowledgments: W.D. Bishop

# Selection Statements

Thus far, our programs execute every statement, sequentially, from top to bottom.

Sometimes we have to make a decision about what to do next.

Selection Statements allow a program to decide what instructions to execute next, based on the current state of the program.

Selection statements are an example of control statements.

# Types of Selection Statements

There are three kinds of selection statement in C++:

1. `if`
2. `if-else`
3. `switch`

We will examine each of these.

Part I

# The `if` Statement

# The `if` Statement

The simplest of the three is the `if` statement.

The basic format of this statement is as follows:

```
if ( condition ) {
    // Statement Block
}
```

If the *condition* is true, then the statement block will execute.

If the *condition* is false, the statement block is skipped.
The statements in that block are not executed.

# Use of the if-Statement

```
#include <iostream>

using namespace std;

int main()
{
    int x;
    cin >> x;

    if ( x >= 25 ) {
        cout << "Condition is true." << endl;
    }
    cout << "Program Finished." << endl;
    return 0;
}
```

[In-Class Demo: the output of this program]

# The `if` Statement

A condition may be a boolean variable, or it can be an expression that evaluates to `true` or `false`.

It may be a simple condition expression ( `x > 0` ) or a more complex one ( `y < 100 && z > 0` ).

Remember that the condition will be short-circuit evaluated.

A more complicated example follows on the next slide:

# Use of the if-Statement

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

int main() {
    int output = 0;
    int input1;
    int input2;

    cout << "Enter Input 1: ";
    cin >> input1;
    cout << "Enter Input 2: ";

    if ( (input1 == 0 && input2 == 1) || ( input1 == 1 && input2 == 0 ) ) {
            output = 1;
    }
    cout << "Output = " << output << endl;
    return 0;
}
```

[In-Class Demo: the output of this program]

# if Statement Pitfall

There is a potential pitfall to the if statement. Use of the { } braces for the statement block following the if statement is technically optional.

This is not a syntax error:

```
if (x > 0)
  y = 1;
```

The statement `y = 1;` is executed only if the condition `x > 0` is true.

If later we edit this code and add the following:

```
if (x > 0)
  y = 1;
  z = 2;
```

This is a potential source of error, but why?

# if Statement Pitfall

```
if (x > 0)
  y = 1;
  z = 2;
```

`z = 2;` is executed regardless of whether `x` is greater than zero.

That might be what you intended, but it might also be an error.

Solution: always use the { and } braces when writing an if-statement.

# The `if-else` Statement

We can build on the if statement with the `else` keyword.

```
if ( condition ) {
    // Statement Block 1
} else {
    // Statement Block 2
}
```

If the *condition* is true, then statement block 1 will execute; statement block 2 will not execute.

If the *condition* is false, then statement block 2 will execute; statement block 1 will not execute.

# The `if-else` Statement

```
cout << "Enter your age: ";
int age;
cin >> age;

if ( age >= 16 ) {
 cout << "You may take the driving test." << endl;
} else {
 cout << "You are not old enough. Sorry." << endl;
}
```

# Use of the if-Statement

```
int main() {
    int output = 0;
    int input1;
    int input2;

    cout << "Enter Input 1: ";
    cin >> input1;
    cout << "Enter Input 2: ";

    if ( input1 == 0 ) {
        output = 1;
    }

    if ( input1 != 0 && input2 == 1 ) {
        output = 2;
    }
    cout << "Output = " << output << endl;
    return 0;
}
```

[In-Class Demo: the output of this program]

# Comments on the Previous Program

You may have noticed some redundancy in the previous program:

Two if conditions, one with `input == 0` and one with `input != 0`.

We have the else-if statement to deal with this situation.

# Use of the if-Statement

```
int main() {
    int output = 0;
    int input1;
    int input2;

    cout << "Enter Input 1: ";
    cin >> input1;
    cout << "Enter Input 2: ";

    if ( input1 == 0 ) {
        output = 1;
    } else if ( input2 == 1 ) {
        output = 2;
    }
    cout << "Output = " << output << endl;
    return 0;
}
```

[In-Class Demo: the output of this program]

# Use of if, else if, else

We always have to start with an if statement.

Zero or more “else if” statements can be added on.

At the end, we may optionally put the else statement.

What if some of the conditions are the same?

# Mutually Exclusive If/Else-If

```
int main() {
    int output = 0;
    int input1;
    int input2;

    cout << "Enter Input 1: ";
    cin >> input1;
    cout << "Enter Input 2: ";

    if ( input1 == 0 ) {
        output = 1;
    } else if ( input1 == 0 ) {
        output = 2;
    }
    cout << "Output = " << output << endl;
    return 0;
}
```

[In-Class Demo: the output of this program]

# Mutually Exclusive

Only one of the blocks will execute; they are all mutually exclusive.
In fact, the condition of the second block will not be evaluated.
Another example of short-circuit evaluation.

In the previous slide, we had two checks of `input1 == 0`.

The first one encountered resolves to true and that block executed.
`output` receives a value of 1.

The next statement executed is the `Console.WriteLine` statement.

# Nested if Statements

It is certainly permitted to have nested if statements.

```
if ( x > 0 ) {
    if ( y < 100 ) {
        output = 7;
    } else {
        output = 10;
    }
}
```

Writing a condition in this way may be clearer than having a lot of else if statements.

There is no effective limit on how many nested if statements you can have, but sometimes it is sensible to combine them for clarity.

Part II

# The `switch` Statement

# The `switch` Statement

The switch statement evaluates a single variable against a large range of alternatives and selects which statement block to execute.

```
switch ( selector )
{
    case label1:
        // Statement block 1
        break;
    case label2:
        // Statement block 2
        break;
    default:
        // Default statement block
        break;
}
```

There can be as many cases as we like.

# The `switch` Statement

The *selector* must be an expression that evaluates to one of:
`bool`, `char`, `int`, or `string`.

The value of the selector is compared to whatever comes after the keyword `case` (e.g., `label1`).

If they are equal, that statement block is executed.

The statement `break;` is used to indicate the end of that option's statement block.

If none of the case options match the selector, the `default` statement block is executed.

# The switch Example

```
char keystroke;
cin >> keystroke;

switch( keystroke )
{
    case 'A':
        Console.Write( "1" );
        break;
    case 'B':
        Console.Write( "2" );
        break;
    case 'C':
        Console.Write( "3" );
        break;
    default:
        Console.Write( "0" );
        break;
}
```

# Analysis of Previous Example

It would be an error if we had two cases labelled with `'A'`.

Like the `if` statement, the cases are mutually exclusive.
Although for `switch`, this is enforced by the compiler.

The `break;` statement is needed to indicate the end of an option.

The `default` block executes if the input didn't match `'A'`, `'B'`, or `'C'`.

# The Default Block

The `default` block executes if the input matches none of the labels.

However, the `default` block is optional; it doesn't have to appear.

If it is not present, and the input does not match any of the labels, none of the blocks will execute. In other words: nothing happens.

# Multiple-Case Example

To avoid having to copy and paste code, you can associate a block of statements with multiple labels.

```
char keystroke = (char) Console.ReadKey( );
switch( keystroke )
{
    case 'a':
    case 'A':
        Console.Write( "1" );
        break;
}
```

# Remember to Take a break

This, however, is valid syntax, but a source of many errors in C++:

```
char keystroke = (char) Console.ReadKey( );
switch( keystroke )
{
    case 'a':
        Console.Write( "7" );
    case 'A':
        Console.Write( "1" );
        break;
}
```

The `break;` statement is missing after case `'a'`.

This would mean that the output would be "71".

# Remember to Take a `break`

The concept of going on from one case to the one below is called "fall through".

The designers of C#, for example, recognized that this was a common source of programmer error.

They therefore chose to explicitly forbid it. But it is a problem in C++.

# `switch` vs `if`

The `switch` and `if` statements are two different ways to represent the same idea: selection statements.

The `switch` statement can be rewritten as an `if-else` statement.

Let's see an example of `switch` and its equivalent `if`.

# Making the switch

```
int switchExpression = 3;
switch (switchExpression)
{
    case 0:
    case 1:
        cout << "Case 0 or 1" << endl;
        break;
    case 2:
        cout << "Case 2" << endl;
        break;
     // 7 - 4 in the following line evaluates to 3.
     case 7 - 4:
         cout << "Case 3" << endl;
         break;
     default:
         cout << "Default case (optional)" << endl;
         break;
}
```

# Making the switch

```
int switchExpression = 3;

if ( switchExpression == 0 || switchExpression == 1 ) {
    cout << "Case 0 or 1" << endl;
} else if ( switchExpression == 2 ) {
    cout << "Case 2" << endl;
} else if ( switchExpression == (7 - 4) ) {
    cout << "Case 3" << endl;
} else {
    cout << "Default case (optional)" << endl;
}
```

# When to Use `if` vs `switch`

You can use the `if-else` statement to replace a `switch` statement so the `if` statement is always applicable.

The `switch` statement may be better when repeatedly checking the value of a single variable.