Conditional and Control Statements in C++

Dr Ian Cornelius

Hello

Hello (1)

Learning Outcomes

1. Understand how to use conditional and control statements in C++
2. Demonstrate the ability to use conditional and control statements in C++

Conditional Statements

Conditional Statements (1)

- **Recap**:
  - A basic form of making a decision using a selection structure
    - the result will return either `1` (true) or `0` (false)
- C++ allows the following types of conditional statementS:
  - `if`
  - `if ... else ...`
  - `if ... else if ....`
  - nested `if ... else ...`
- These statements are structured slightly different compared to Python

`if` Statements

`if` Statements (1)

- Often referred to as a decision-making statement
- Used to control the flow of execution for statements and to test an expression
  - tests logically whether a condition is `true` or `false`
- **Note**: Unlike Python the comparison expression is wrapped in brackets (`()`)
  - there are also curly brackets (`{}`) which enclose the return statement

```
if (variable == value) {
    ...
}
```

`if` Statements (2)

Example: `if` Statement i

- Declare a variable `ifExample1` to store the integer value `1`
- Perform a comparison check: `iseXample1` equal to `1`?
  - if `true`: print the value `True, ifExample1 is 1` to the screen

```cpp
#include <iostream>
int ifExample1 = 1;
int main() {
  if (ifExample1 == 1) {
    std::cout << "True, ifExample1 is 1" << std::endl;
  }
  return 0;
}
```

```
True, ifExample1 is 1
```

`if` Statements (3)

Example: `if` Statement ii

- Declare a variable `ifExample1` to store the integer value `2`
- Perform a comparison check: `ifExample1` equal to `1`?
  - if `True`: print the value `True` to the screen
  - otherwise, continue executing the code

```
#include <iostream>
int ifExample1 = 2;
int main() {
  if (ifExample1 == 1) {
    std::cout << "True, ifExample1 is 1" << std::endl;
  }
  std::cout << "Outside the 'if' statement." << std::endl;
  return 0;
}
```

```
Outside the 'if' statement.
```

`if else` Statements

`if else` Statements (1)

- Known as an alternative execution, whereby there are two possibilities
  - the condition statement determines which of the two statements gets executed
- The `else` is used as the ultimate result for a test expression
  - this result is only met if all other statements are `false`

```
if (variable == value) {
  ...
} else {
  ...
}
```

`if else` Statements (2)

Example: `if else` Statement i

- Declare a variable `ifExample1` to store the integer value `1`
- Perform a comparison check: `ifExample1` equal to `1`?
  - if `True`: print the value `True, ifExample1 is 1` to the screen
  - otherwise, print `False, ifExample1 is not 1`

```
#include <iostream>
int ifExample1 = 1;
int main() {
  if (ifExample1 == 1) {
    std::cout << "True, ifExample1 is 1" << std::endl;
  } else {
    std::cout << "False, ifExample1 is not 1" << std::endl;
  }
  return 0;
}
```

```
True, ifExample1 is 1
```

`if else` Statements (3)

Example: `if else` Statement ii

- Declare a variable `ifExample1` to store the integer value `2`
- Perform a comparison check: `ifExample1` equal to `1`?
  - if `True`: print the value `True, ifExample1 is 1` to the screen
  - otherwise, print `False, ifExample1 is not 1`

```
#include <iostream>
int ifExample1 = 2;
int main() {
  if (ifExample1 == 1) {
    std::cout << "True, ifExample1 is 1" << std::endl;
  } else {
    std::cout << "False, ifExample1 is not 1" << std::endl;
  }
  return 0;
}
```

`else if` Statements

`else if` Statements (1)

- Evaluates two or more possibilities from a collection of comparison statements
- The condition allows for two or more possibilities, known as a **chained conditional**

```
if (variable > value) {
  ...
} else if (variable < value) {
  ...
} else {
  ...
}
```

`else if` Statements (2)

Example: `else if` Statement i

- Declare a variable `ifExample1` to store the integer value `1`
- Declare a variable `ifExample2` to store the integer value `3`
- Perform a comparison check: `ifExample1` equal to `1`?
  - if `True`: print the value `[True] ifExample1 is 1` to the screen
- **Stop the comparison checks!**

```cpp
#include <iostream>
int ifExample1 = 1;
int ifExample2 = 3;
int main() {
  if (ifExample1 == 1) {
    std::cout << "[True] ifExample1 is 1" << std::endl;
  } else if (ifExample2 == 2) {
    std::cout << "[True] ifExample2 is 2" << std::endl;
  } else {
    std::cout << "[False] ifExample1 is not 1, and ifExample2 is
  }
  return 0;
```

```
[True] ifExample1 is 1
```

`else if` Statements (3)

Example: `else if` Statement ii

- Declare a variable `ifExample1` to store the integer value `3`
- Declare a variable `ifExample2` to store the integer value `2`
- Perform a comparison check: `ifExample1` equal to `1`?
  - if `True`: print the value `[True] ifExample1 is 1` to the screen
  - otherwise, perform another conditional check: `ifExample2` is equal to `2`?
    - if `True`: print the value `[True] ifExample2 is 2` to the screen
- **Stop the comparison checks!**

```cpp
#include <iostream>
int ifExample1 = 3;
int ifExample2 = 2;
int main() {
  if (ifExample1 == 1) {
    std::cout << "[True] ifExample1 is 1" << std::endl;
  } else if (ifExample2 == 2) {
    std::cout << "[True] ifExample2 is 2" << std::endl;
  } else {
    std::cout << "[False] ifExample1 is not 1, and ifExample2 is
  }
  return 0;
```

```
[True] ifExample2 is 2
```

`else if` Statements (4)

Example: `else if` Statement iii

- Declare a variable `ifExample1` to store the integer value `1`
- Declare a variable `ifExample2` to store the integer value `3`
- Perform a comparison check: `ifExample1` equal to `1`?
  - if `True`: print the value `[True] ifExample1 is 1` to the screen
  - otherwise, perform another conditional check: `ifExample2` is equal to `2`?
    - if `True`: print the value `[True] ifExample2 is 2` to the screen
    - otherwise, print `[False], ifExample1 is not 1, and ifExample2 is not 2`

```
#include <iostream>
int ifExample1 = 1;
int ifExample2 = 3;
int main() {
  if (ifExample1 == 1) {
    std::cout << "[True] ifExample1 is 1" << std::endl;
  } else if (ifExample2 == 2) {
    std::cout << "[True] ifExample2 is 2" << std::endl;
  } else {
    std::cout << "[False] ifExample1 is not 1, and ifExample2 is
  }
  return 0;
```

```
[True] ifExample1 is 1
```

Nested `if` Statements

Nested `if` Statements (1)

- `if` statements can be written inside each other
  - this is known as **nesting**

```
if (variable == value) {
  if (variable1 == value1) {
    ...
  } else if (variable1 == value2) {
    ...
  } else {
    ...
  }
} else {
  if (variable2 == value1) {
    ...
  } else {
```

Nested `if` Statements (2)

Example: Nested `if` Statement i

- Declare a variable `ifExample1` to store the integer value `1`
- Declare a variable `ifExample2` to store the integer value `2`
- Perform a comparison check: `ifExample1` equal to `1`?
  - if `True`: perform another comparison check: `ifExample2` equal to `2`?
    - if `True`: print `[True] ifExample1 is 1 and ifExample2 is 2`
- **Stop the comparison checks!**

```
#include <iostream>
int ifExample1 = 1;
int ifExample2 = 2;
int main() {
  if (ifExample1 == 1) {
    if (ifExample2 == 2) {
      std::cout << "[True], ifExample1 is 1, and ifExample2 is 2
    } else if (ifExample2 == 4) {
      std::cout << "[True] ifExample1 is 1 and ifExample2 is 4" <
    } else {
      std::cout << "[True] ifExample1 is 1 but, ifExample2 is no
    }
```

```
[True], ifExample1 is 1, and ifExample2 is 2
```

Nested `if` Statements (3)

Example: Nested `if` Statement ii

- Declare a variable `ifExample1` to store the integer value `1`
- Declare a variable `ifExample2` to store the integer value `4`
- Perform a comparison check: `ifExample1` equal to `1`?
  - if `True`: perform another comparison check: `ifExample2` equal to `2`?
    - if `True`: print `[True] ifExample1 is 1 and ifExample2 is 2`
    - otherwise perform another comparison check: `ifExample2` equal to `4`?
      - if `True`: print `[True] ifExample1 is 1 and ifExample2 is 4`
- **Stop the comparison checks!**

```cpp
#include <iostream>
int ifExample1 = 1;
int ifExample2 = 4;
int main() {
  if (ifExample1 == 1) {
    if (ifExample2 == 2) {
      std::cout << "[True], ifExample1 is 1, and ifExample2 is 2
    } else if (ifExample2 == 4) {
      std::cout << "[True] ifExample1 is 1 and ifExample2 is 4" <
    } else {
      std::cout << "[True] ifExample1 is 1 but, ifExample2 is no
    }
```

```
[True] ifExample1 is 1 and ifExample2 is 4
```

Nested `if` Statements (4)

Example: Nested `if` Statement iii

- Declare a variable `ifExample1` to store the integer value `2`
- Declare a variable `ifExample2` to store the integer value `5`
- Perform a comparison check: `ifExample1` equal to `2`?
  - if `True`: perform another comparison check: `ifExample2` equal to `2`?
  - otherwise, print `[False] ifExample1 is not 1`
- **Stop the comparison checks!**

```cpp
#include <iostream>
int ifExample1 = 2;
int ifExample2 = 5;
int main() {
  if (ifExample1 == 1) {
    if (ifExample2 == 2) {
      std::cout << "[True], ifExample1 is 1, and ifExample2 is 2
    } else if (ifExample2 == 4) {
      std::cout << "[True] ifExample1 is 1 and ifExample2 is 4" <
    } else {
      std::cout << "[True] ifExample1 is 1 but, ifExample2 is no
    }
```

```
[False] ifExample1 is not 1
```

Control Statements

Control Statements (1)

- Typically, statements in code will be executed sequentially
- There are some situations that require a block of code to be repeated
  - i.e. summing numbers, capturing multiple user-input etc.
- Control statements, otherwise known as loop statements, are required
- Three types of loops in C++:
  - `while`
  - `do ... while ...`
  - `for`

`while` Loops

`while` Loops (1)

- A loop that executes zero or more times before it is terminated
- Used to evaluate upon a condition
  - if the condition evaluates to `1` (`true`) the code inside the loop will be executed
  - if the condition evaluates to `0` (`false`) the loop will terminate

```
while (variable < value) {
  ...
  variable += 1;
}
```

`while` Loops (2)

- Initiate a new variable: `whileExample1 = 0`
- Provide a condition to evaluate on: `whileExample1 <= 5` evaluates to `1` (`true`)
  - execute the code inside the while loop:
    - prints the value of `whileExample1`
    - increments `whileExample1` by `1`
- Loop is repeated until condition evaluates to `0` (`false`)

```
#include <iostream>
int whileExample1 = 0;
int main() {
  while (whileExample1 <= 5) {
    std::cout << "whileExample1 -> " << whileExample1 << std::endl
    whileExample1 += 1;
  }
  return 0;
}
```

```
whileExample1 -> 0
whileExample1 -> 1
whileExample1 -> 2
whileExample1 -> 3
whileExample1 -> 4
whileExample1 -> 5
```

`while` Loops (3)

Breaking a `while` Loop

- `break` statements can be used to stop the loop if a condition is evaluated to `true`
- Initiate a new variable: `whileExample1 = 0`
- Provide a condition to evaluate on: `whileExample1 <= 5` evaluates to `1` (`true`)
  - execute the code inside the while loop:
    - prints the value of `whileExample1`
    - increments `whileExample1` by `1`
- Loop is repeated until the conditional statement in the loop evaluates to `1` (true)
  - in this instance, when `whileExample1` is `2`

```cpp
#include <iostream>
int whileExample1 = 0;
int main() {
  while (whileExample1 <= 5) {
    std::cout << "whileExample1 -> " << whileExample1 << std::endl
    if (whileExample1 == 2) {
      break;
    }
    whileExample1 += 1;
  }
  return 0;
}
```

```
whileExample1 -> 0
whileExample1 -> 1
whileExample1 -> 2
```

`while` Loops (4)

Skipping an Iteration

- `continue` statements can stop the current iteration and continue onto the next
- Initiate a new variable: `whileExample1 = 0`
- Provide a condition to evaluate on: `whileExample1 <= 5` evaluates to `1` (`true`)
  - execute the code inside the while loop:
    - prints the value of `whileExample1`
    - increments `whileExample` by `1`
- Loop is repeated until the conditional statement in the loop evaluates to `1` (true)
  - in this instance, when `whileExample1` is `2`

```cpp
#include <iostream>
int whileExample1 = 0;
int main() {
  while (whileExample1 <= 5) {
    whileExample1 += 1;
    if (whileExample1 == 2) {
      std::cout << "SKIPPED" << std::endl;
      continue;
    }
    std::cout << "whileExample1 -> " << whileExample1 << std::endl
  }
  return 0;
```

```
whileExample1 -> 1
SKIPPED
whileExample1 -> 3
whileExample1 -> 4
whileExample1 -> 5
whileExample1 -> 6
```

`while` Loops (5)

Infinite `while` Loops

- Infinite loops can be constructed by using a `true` value after the `while` keyword
  - in this case with C++ - `1`
- Will continue incrementing `whileExample1` until it reaches a certain value
  - in this instance `whileExample1` must be equal to `5`
- If there is no condition to check in the loop, it will continue incrementing

```
#include <iostream>
int whileExample1 = 0;
int main() {
  while(1) {
    std::cout << "whileExample1 -> " << whileExample1 << std::endl
    whileExample1 += 1;
    if (whileExample1 == 5) {
      break;
    }
  }
  return 0;
}
```

```
whileExample1 -> 0
whileExample1 -> 1
whileExample1 -> 2
whileExample1 -> 3
whileExample1 -> 4
```

`do ... while` Loops

`do ... while` Loops (1)

- A variant of the `while` loop structure
- One important difference:
  - the execution of a `do ... while` is performed before the conditional check is evaluated

```
do {
  ...
}
while (condition);
```

`do ... while` Loops (2)

- Initiate a new variable: `doWhileExample1 = 0`
- Execute the code inside the `do` statement:
  - prints the value of `doWhileExample1`
  - increments `doWhileExample1` by `1`
- `doWhileExample1 <= 5` evaluates to `1` (`true`)
- Loop is repeated until condition evaluates to `0` (`false`)

```
#include <iostream>
int doWhileExample1 = 0;
int main() {
  do {
    std::cout << "doWhileExample1 -> " << doWhileExample1 << std::
    doWhileExample1 += 1;
  }
  while (doWhileExample1 <= 5);
  return 0;
}
```

```
doWhileExample1 -> 0
doWhileExample1 -> 1
doWhileExample1 -> 2
doWhileExample1 -> 3
doWhileExample1 -> 4
doWhileExample1 -> 5
```

`for` Loops

`for` Loops (1)

- A loop that is designed to increment a counter over a given range of values
- They are best suited for problems that need to iterate a specific number of times
  - i.e. looping through a directory or set of files
- Considered to be a **pre-test** loop
  - they check their condition before execution
- `for` loops are useful because...
  - they know the number of times a loop should be iterated
  - they use a counter
  - require a *false* condition to terminate the loop

```
for (initialisation; condition; update) {
  ...
}
```

- `initialisation`: initialises the counter-variable
  - i.e. `int i = 0;`
- `condition`: if `1` (true) the body of the loop is executed, if `0` (false) the loop is terminated
- `update`: increments the counter-variable and checks the `condition` again
  - i.e. `i++`

`for` Loops (2)

Example: Iterating Forwards

- Initialise our counter, `int i = 0;`
- Provide a conditional check:
  - i.e. `i < 5` - checks whether the integer is less than `5`
- If the condition is `1` (true) then execute the code within the `for` body
  - in this instance, it will print the value of `i`
- Increment the counter by one, `i++`
- Loop until the conditional check evaluates to `0` (false)

```cpp
#include <iostream>
int main() {
  for (int i = 0; i < 5; i++) {
    std::cout << "i -> " << i << std::endl;
  }
}
```

```
i -> 0
i -> 1
i -> 2
i -> 3
i -> 4
```

`for` Loops (3)

Example: Iterating Backwards

- Initialise our counter, `int i = 5;`
- Provide a conditional check:
  - i.e. `i > 0` - checks whether the integer is less than `5`
- If the condition is `1` (true) then execute the code within the `for` body
  - in this instance, it will print the value of `i`
- Increment the counter by one, `i--`
- Loop until the conditional check evaluates to `0` (false)

```cpp
#include <iostream>
int main() {
  for (int i = 5; i > 0; i--) {
    std::cout << "i -> " << i << std::endl;
  }
}
```

```
i -> 5
i -> 4
i -> 3
i -> 2
i -> 1
```

`for` Loops (4)

Range-Based Loops

- Range-based loops can be used to work on arrays and vectors
- The syntax of a range-based loop:

```
for (variable : [array or vector]) {
  ...
}
```

- For each element in the array or vector, the loop is executed
  - the element will be assigned to the variable

`for` Loops (5)

Example: Range-Based Loops i

- Create the variable to store an element from the array
  - i.e. `int i`
- Each element inside the array will be assigned to `i`
  - it will then be printed to the terminal window
- Loop will terminate when no more elements exist in the array

```
#include <iostream>
int intArrayExample1[5] = {0, 1, 2, 3, 4};
int main() {
  for (int i : intArrayExample1) {
    std::cout << "i -> " << i << std::endl;
  }
}
```

```
i -> 0
i -> 1
i -> 2
i -> 3
i -> 4
```

`for` Loops (6)

Example: Range-Based Loops ii

- Create the variable to store an element from the array
  - i.e. `auto item`
- Each element inside the map will be assigned to `item`
  - the key and value of each element can be accessed by:
    - `first`: returns the key
    - `second`: returns the value
  - they then be printed to the terminal window
- Loop will terminate when no more elements exist in the map

```
#include <iostream>
#include <map>
std::map<int, std::string> mapExample1 = {{0, "Ian Cornelius"}, {1
int main() {
    for (auto &item : mapExample1) {
        std::cout << "item.first -> " << item.first << std::endl;
        std::cout << "item.second -> " << item.second << std::endl
    }
}
```

```
item.first -> 0
item.second -> Ian Cornelius
item.first -> 1
item.second -> Terry Richards
item.first -> 2
item.second -> Daniel Goldsmith
```

`for` Loops (7)

Infinite Loops

- Infinite loops can also be created using a `for`
- A condition is required that will always evaluate to `1` (otherwise known as `true`)

```
for(int i = 1; i > 0; i++) {
  ...
}
```

- In this instance, the integer `i` will always be greater than `0`
- Some method of terminating will be required
  - i.e. checking whether `i` is a particular value

Goodbye

Goodbye (1)

Questions and Support

- Questions? Post them on the **Community Page** on Aula
- Additional Support? Visit the Module Support Page
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk