Introductory Robot Programming

ENPM809Y

Reading Material 02 – Program Flow Control

Zeid Kootbally
zeidk@umd.edu

Fall 2020

Overview I

Overview II

Constructs
`for` Loop
Range-based `for` Loop
`while` Loop
`do-while` Loop
Nested Loops

List of Exercises

Flow Control

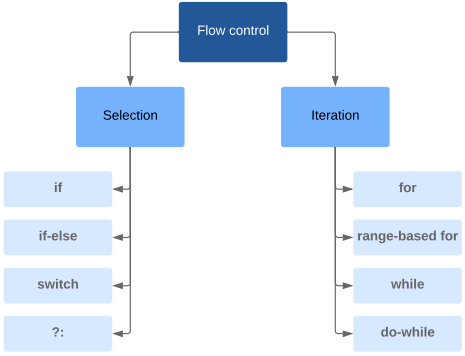
- So far we have learned how to write programs that run sequentially.

```
int main(){
    int a{2}, b{1};
    std::cout << "a + b: " << a + b << std::endl;
    std::cout << "a - b: " << a - b << std::endl;
}
```

- Programs are not limited to a linear sequence of statements.
- During its process, a program may repeat segments of code, or take decisions and bifurcate.

Flow Control

- For that purpose, C++ provides flow control statements that serve to specify what has to be done by our program, when, and under which circumstances.



Flow Control | Statements

- Many of the flow control statements explained in this lecture require a generic (sub)statement as part of its syntax. This statement may either be:
  - A simple statement, terminated with a semicolon, e.g., `int x{3};`
  - A compound statement: A group of statements (each of them terminated by its own semicolon), but all grouped together in a block, enclosed in curly braces.

```
int main(){
    {//--Start of compound statement 1
        int a{10};
        if (a <= 100)
        {//--Start of compound statement 2
            std::cout << "a is: " << a << std::endl;
        }//--End of compound statement 2
    }//--End of compound statement 1
}
```

  - The entire block is considered a single statement (composed itself of multiple substatements).

Selection

- Selection is used for decisions or branching (choosing between 2 or more alternative paths).
- The different types of selection statements in C++ are:
  - `if`
  - `if-else`
  - `switch`
  - `?:` conditional operator (similar to `if-else`)

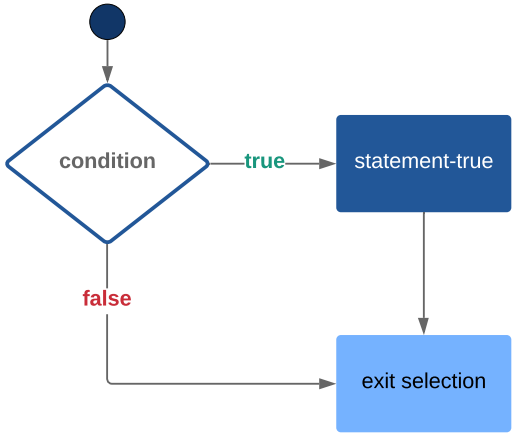
Selection | `if` Statement

- Conditionally executes another statement.
- Used where code needs to be executed based on a condition.

```
if (condition) statement-true
```

- `condition`: Expression which is contextually convertible to `bool`.
- `statement-true`: Any statement (often a compound statement), which is executed if `condition` evaluates to `true`.
  - If `condition` is `true`, `statement-true` is executed.
  - If `condition` is `false`, `statement-true` is simply ignored and the program continues right after the entire selection statement.

Selection | `if` Statement



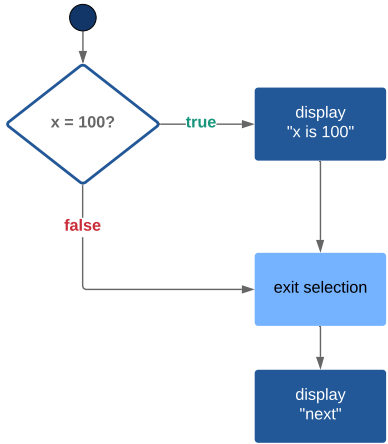
Selection | `if` Statement

```
int x{10};
if (x == 100)
    std::cout << "x is 100\n";
std::cout << "next\n";
```

```
next
```

```
int x{100};
if (x == 100)
    std::cout << "x is 100\n";
std::cout << "next\n";
```

```
x is 100
next
```



Selection | `if` Statement

- Example of a compound statement for the `if` statement.

```
int main(){
    int x{10};

    if (x == 100){//--start of the compound statement
        std::cout << "x is 100";
        std::cout << std::endl;
    }//--end of the compound statement
}
```

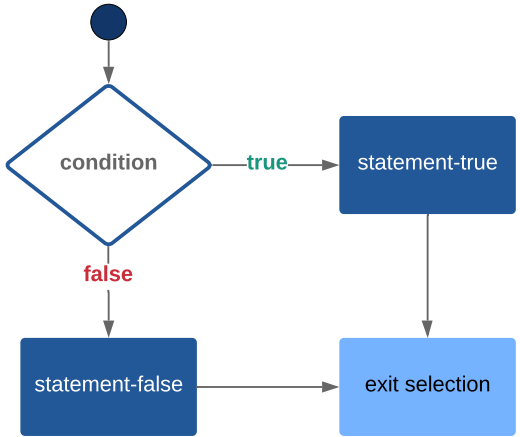
Selection | `if-else` Statement

- Selection statements with `if` can also specify what happens when the condition is not fulfilled with the `else` keyword.

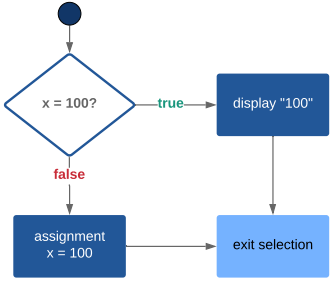
```
if (condition) statement-true else statement-false
```

  - `condition`: Expression which is contextually convertible to `bool`.
  - `statement-true`: Any statement (often a compound statement), which is executed if `condition` evaluates to `true`.
  - `statement-false`: Any statement (often a compound statement), which is executed if `condition` evaluates to `false`.
    - If `condition` is `true`, `statement-true` is executed.
    - If `condition` is `false`, `statement-false` is executed.

Selection | `if-else` Statement



Selection | `if-else` Statement




```
int x{100};
if (x == 100)//-- condition
    std::cout << "100" << std::endl;//--statement-true
else
    x = 100;//--statement-false
```

Selection | `if-else` Statement

- Just like the `if` statement, the `else` statement can also have its own block statement.

```
if (x == 100){
    std::cout << "Display x" << std::endl;
    std::cout << "x is 100" << std::endl;
}
else{
    std::cout << "Set x to 100" << std::endl;
    x = 100;
}
```

Selection | `if-else` Statement

- In many cases, we can group `if else` statements together.
- Example: Display students' letter grade given their score on an exam.

```
int score{80};

if (score >= 90)
    std::cout << "A\n";
else if (score >= 80)
    std::cout << "B\n";
else if (score >= 70)
    std::cout << "C\n";
else if (score >= 60)
    std::cout << "D\n";
else // all others must be F
    std::cout << "F\n";
```

Selection | Pseudo Code

- Pseudo code:
  - An informal high-level description of the operating principle of a computer program or other algorithm.
  - It uses the structural conventions of a normal programming language, but is intended for human reading rather than machine reading.
  - You can reuse the pseudocode to implement the program or algorithm in other programming languages.
  - *Best practice*: Always write the pseudocode of a program (on paper if possible) before you write the code in a programming language.

Selection | Exercise #1

- Write the pseudo code for the following program (solutions provided in slides 20 and 21)
- Write a program which:
  1. Prompts the user to enter 3 integer numbers, where each number represents a side of a triangle.
  2. Outputs the type of the triangle (equilateral, isosceles, or scalene).
     - **Equilateral triangle**: All sides equal in length.
     - **Isosceles triangle**: Two sides of equal length.
     - **Scalene triangle**: All sides unequal in length.
- Example of outputs.

```
Enter the length of each side of the triangle:  20 34 34
This is an isosceles triangle
```

Selection | Exercise #1

- Here is an example of pseudo code for Exercise #1.

  Data: Three sides of a triangle
  Result: Type of the triangle

  ```
  Prompt user to enter the length for each side;
  if all sides are equal then
      Equilateral;
  else
      if two sides are equal then
          Isosceles;
      else
          Scalene;
      end
  end
  ```

Selection | Exercise #1

- Here is another example of pseudo code for Exercise #1.

  Data: Three sides of a triangle
  Result: Type of the triangle

  Prompt user to enter the length for each side: $s1$,$s2$,$s3$;
  if $s1$=$s2$=$s3$ **then**
  | Equilateral;
  else
  | **if** $s1$=$s2$ or $s1$=$s3$ or $s2$=$s3$ **then**
  | | Isosceles;
  | **else**
  | | Scalene;
  | **end**
  end

Selection | Nested if Statements

- Many times, the logic we need to solve a problem is more complex than an `if-else` statement.
- We may need several layers of logic.
- Since an `if` or `if-else` statement is a statement, we can use it in C++ anywhere a statement is legal.
- In the code below we replaced the `statement` in the left code snippet with an `if else` statement.

```
if (condition)
    statement
```

```
if (condition)
    if (condition)
        statement-true
    else
        statement-false
```

```
int score{93};
if (score >= 93) {
    if (score > 96)
        std::cout << "A+\n";
    else
        std::cout << "A\n";
}
else std::cout << "No grade A!\n";
```

Selection | Exercise #2

- This exercise involves:
  - 1 robot

  - 3 tables (blue, purple, and green tables)

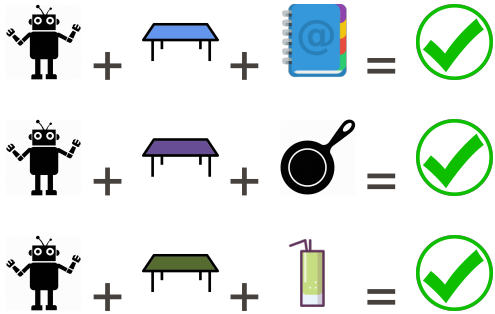
  - 3 objects (book, pan, and glass)

Selection | Exercise #2

- The robot can walk to each table.
- A human places each object on a table.
- Each table can contain only one of the three objects.
- Rules: The robot can pick up:
  - book if the robot is in front of the blue table and the book is on the blue table,
  - pan if the robot is in front of the purple table and the pan is on the purple table,
  - glass if the robot is in front of the green table and the glass is on the green table.

Selection | Exercise #2

- 3 successful combinations:



Selection | Exercise #2

- Examples of failed combinations:

Selection | Exercise #2

- Start writing the pseudo code for the instructions below.
  - Using nested `if` statements, write a program which:
    1. Tells the robot to move to a table,
    2. Prompts the user for the object to place on that table (from the previous step),
    3. Displays a message on the screen whether or not the robot can pick up the object.

```
Which table should the robot go to? (b)lue/(p)urple/(g)reen: g
Which object is on that table? (b)ook/(p)an/(g)lass: g
Sucess: The robot picks up the glass from the green table.
```

```
Which table should the robot go to? (b)lue/(p)urple/(g)reen: b
Which object is on that table? (b)ook/(p)an/(g)lass: g
Failure: The robot cannot pick up the glass from the blue table.
```

- Note: Check the user inputs are valid, i.e, they have to be either `b`, `p`, or `g`.

Selection | `switch` Statement

- A `switch` statement can replace multiple `if` or `if-else` checks.
- A `switch` statement tests the value of a variable and compares it with multiple cases. Once the case match is found, a block of statements associated with that particular case is executed.
- The overall idea behind `switch` statements is simple: the `switch` expression is evaluated to produce a value, and each case label is tested against this value for equality. If a case label matches, the statements after the case label are executed. If no case label matches the `switch` expression, the statements after the default label are executed (if it exists).
- Because of the way they are implemented, `switch` statements are typically more efficient than `if-else` chains.

Selection | `switch` Statement

`switch (condition) statement`

- We start a `switch` statement by using the `switch` keyword, followed by the `condition` that we would like to evaluate. Typically this `condition` is just a single variable, but it can be something more complex.
- The one restriction on this `condition` is that it must evaluate to an integral type (`char`, `short`, `int`, `long`, `long long`, or `enum`).
- Floating point variables, strings, and other non-integral types may not be used here.
- Following the `switch` expression, we declare a `statement` (typically a compound `statement`). Inside the `statement` we use labels to define all of the values we want to test for equality. There are two kinds of labels: `case` and `default`

Selection | switch Statement | if-else Vs. switch

```cpp
enum Color{
    kBlack,
    kWhite,
};
//--With if-else statements
void PrintColor(Color color){
    if (color == kBlack)
        std::cout << "Black";
    else if (color == kWhite)
        std::cout << "White";
    else
        std::cout << "Unknown";
}

int main(){
    Color my_color{kBlack};
    PrintColor(my_color);
}
```

```cpp
enum Color{
    kBlack,
    kWhite,
};
//--With if-else statements
void PrintColor(Color color){
    switch (color) {
        case kBlack:
            std::cout << "Black";
            break;
        case kWhite:
            std::cout << "White";
            break;
        default:
            std::cout << "Unknown";
    }
}

int main(){
    Color my_color{kBlack};
    PrintColor(my_color);
}
```

Selection | `switch` Statement | `case` Labels

- The first kind of label is the `case` label, which is declared using the `case` keyword and followed by a constant expression.
  - A constant expression is one that evaluates to a constant value. In other words, either a literal (such as 5), an `enum` (such as `kWhite`), or a constant variable (such as `x`, when `x` has been defined as a `const int`).
- The constant expression following the `case` label is tested for equality against the expression following the `switch` keyword. If they match, the code under the `case` label is executed.
- Note: All `case` label expressions must evaluate to a unique value. The following code is not valid:

```
switch (x){
    case 1:
    case 1:  //--Illegal -- already used value 1
    case kWhite: //--Illegal, kWhite evaluates to 1
}
```

Selection | switch Statement | case Labels

- It is possible to have multiple case labels refer to the same statements. The following function uses multiple cases to test if the c parameter is A/a/B/b/C/c.

```
bool IsABC(char c){
    switch (c){
        case 'A': // if c is A
        case 'a': // if c is a
        case 'B': // if c is B
        case 'b': // if c is b
        case 'C': // if c is C
        case 'c': // if c is c
            return true; //--Then return true
        default:
            return false;
    }
}
int main(){
    if (IsABC('d'))
        std::cout << " The character is  A/a/B/b/C/c" << std::endl;
}
```

- In the case where c is A/a/B/b/C/c, the first statement after the matching case statement is executed, which is return true; in this example.

Selection | `switch` Statement | `default` Label

- The second kind of label is the `default` label (often called the "default case"), which is declared using the `default` keyword.
- The code under this label gets executed if none of the cases match the `switch` expression.
- The `default` label is optional, and there can only be one `default` label per `switch` statement.
- It is also typically declared as the last label in the `switch` block, though this is not strictly necessary.
- In the `IsDigit()` example from the previous slide, if `c` is NOT A/a/B/b/C/c, the `default` case executes and `return false;` is executed.

Selection | `switch` Statement | Execution and Fall-though

- One of the trickiest things about `case` statements is the way in which execution proceeds when a `case` is matched.
- When a `case` is matched (or the `default` is executed), execution begins at the first statement following that label and continues until one of the following termination conditions is true:
  - The end of the `switch` block is reached.
  - A `return` statement occurs.
  - A `break` statement occurs.
  - Something else interrupts the normal flow of the program (e.g. a call to `exit()`, an exception occurs, etc).

Selection | switch Statement | Execution and Fall-though

- Note: If none of these termination conditions are met, cases will overflow into subsequent cases! You probably do not want the following result.

```
switch (2){
    case 1: //--Does not match
        std::cout << 1 << '\n'; //--Skipped
    case 2: //--Match!
        std::cout << 2 << '\n'; //--Execution begins here
    case 3:
        std::cout << 3 << '\n'; //--This is also executed
    default:
        std::cout << "default case" << '\n'; //--This is also executed
}
```

- Fall-through is almost never desired by the programmer, so in the rare case where it is, it is common practice to leave a comment stating that the fall-through is intentional.

Selection | switch Statement | break Statement

- A `break` statement tells the compiler that we are done with this `switch`. After a `break` statement is encountered, execution continues after the end of the `switch` block.

```
switch (2){
   case 1: //--Does not match -- skipped
       std::cout << 1 << '\n';
       break;
   case 2: //--Match!  Execution begins at the next statement
       std::cout << 2 << '\n'; // Execution begins here
       break; //--Break terminates the switch statement
   case 3:
       std::cout << 3 << '\n';
       break;
   default:
       std::cout << "default case" << '\n';
       break;
}
//--Execution resumes here
```

Selection | switch Statement | Summary

- An example with a robot picking up a peg.

```
enum Pegs{kPegRed, kPegYellow, kPegOrange};//--peg enumerators

int main(){
    Pegs peg{kPegYellow};
    switch (peg){
        case kPegRed:
            std::cout << "Robot picks up the red peg" << std::endl;
            break;
        case kPegYellow:
            std::cout << "Robot picks up the yellow peg" << std::endl;
            break;
        case kPegOrange:
            std::cout << "Robot picks up the orange peg" << std::endl;
            break;
        default:
            std::cout << "Should never execute" << std::endl;
            break;
    }
}
```

Selection | `switch` Statement | Exercise #3

- Write the pseudo code for the following problem.
- Using `switch`, write a program which displays the score to get a letter grade (A,B,C,D, or F) based on the user's input.
  - `A` or `a`, display `You need a 90 or above.`
  - `B` or `b`, display `You need 80-89 for a B.`
  - `C` or `c`, display `You need 70-79 for an average grade.`
  - `D` or `d`, display `You need 60-69.`
  - `F` or `f`, display `Are you sure (Y/N)?`
    - `Y` or `y`, display `It seems you didn't study.`
    - `N` or `n`, display `Good answer, now go study!`
- Illegal letter grade, display `Not a valid grade` and exit.
- Illegal selection for `Are you sure (Y/N)?`, display `Illegal choice` and exit.

Selection | `switch` Statement | Exercise #3

- Example #1

```
Expected final letter grade for this course (A/B/C/D/F)?  A
You need a 90 or above.
```

- Example #2

```
Expected final letter grade for this course (A/B/C/D/F)?  D
You need 60-69.
```

- Example #3

```
Expected final letter grade for this course (A/B/C/D/F)? F
Are you sure (Y/N)?  N
Good answer, now go study!
```

Selection | Conditional Operator

- The conditional operator (`?:`) (also sometimes called the "ternary operator") is a ternary operator (it takes 3 operands).
- The `?:` operator provides a shorthand method for doing a particular type of `if-else` statement.

```
(condition) ? expression1 : expression2;
```

  - `condition` evaluates to a Boolean expression. If `condition` is `true` then the value of `expression_1` is returned, else the value of `expression_2` is returned.
- The following two code snippets are equivalent.

```
//--if-else statements
int x{1}, y{2}, larger{};
if (x > y)
    larger = x;
else
    larger = y;
```

```
//--conditional operator
int x{1}, y{2}, larger{};
larger = ((x > y) ? x : y);
```

Selection | Conditional Operator

- To properly comply with C++ type checking, both expressions in a conditional statement must either match, or the second expression must be convertible to the type of the first expression.
- The following code will not compile. One of the expressions is an integer, and the other is a string literal. The compiler will try to find a way to convert the string literal to an integer, but since it does not know how, it will give an error. In such cases, you will have to use an `if-else` statement.

```
//--conditional operator
int x{1}, y{2};
std::cout << ((x > y) ? x : "y is larger");
```

- *Best practice*: Always parenthesize the conditional part of the conditional operator, and consider parenthesizing the whole thing as well.

Iteration

- Iterations allow the execution of a statement or a compound statement repeatedly.
- Iterations consist of a loop condition and the body that contains the statements to repeat.
- Use cases:
  - a specific number of times.
  - for each element in a collection (e.g., vector).
  - until a condition becomes false.

Iteration | Constructs

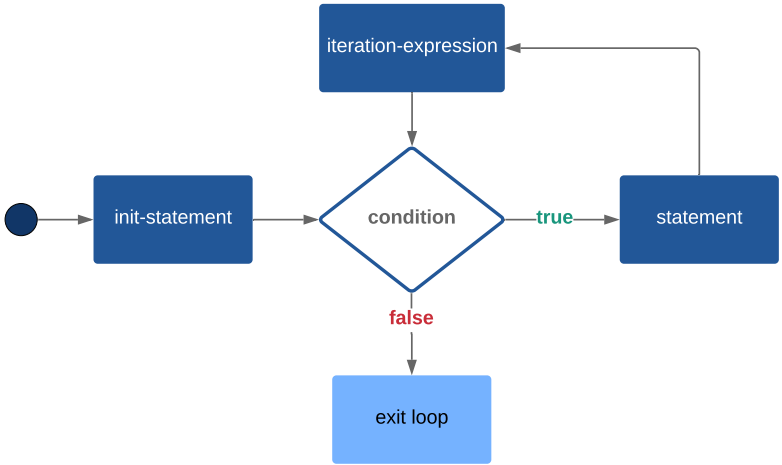
- `for` loop
  - Iterate a specific number of times.
- Range-based `for` loop
  - One iteration for each element in a collection.
- `while` loop
  - Iterate while condition is true.
  - Stops when the condition becomes false.
  - Check the condition at the beginning of the operation.
- `do-while` loop
  - Iterate while condition remains true.
  - Stop when the condition becomes false.
  - Check the condition at the end of the operation.

Iteration | `for` Loop

```
for (init-statement; condition; iteration-expression)
    statement
```

- A `for` statement is evaluated in 3 parts:
  1. The `init-statement` consists of variable definitions and initialization. This statement is only evaluated once, when the loop is first executed.
  2. `condition` is then evaluated. If this evaluates to `false`, the loop terminates immediately. If this evaluates to `true`, the `statement` is executed.
  3. After the `statement` is executed, the `iteration-expression` is evaluated. Typically, this expression is used to increment or decrement the variables declared in the `init-statement`. After the `iteration-expression` has been evaluated, the loop returns to step 2.

Iteration | `for` Loop



Iteration | for Loop | Example

```
for (int i{0}; i < 10; ++i)
    std::cout << i << " ";
```

- First, we declare a loop variable named `i`, and initialize it with the value `0`.
- Second, `i < 10` is evaluated, and since `i` is `0`, `0 < 10` evaluates to `true`. Consequently, the statement executes, which prints `0`.
- Third, `++i` is evaluated, which increments `i` to `1`. Then the loop goes back to the second step.
- `1 < 10` is evaluated to `true`, so the loop iterates again. The statement prints `1`, and `i` is incremented to `2`.
- `2 < 10` evaluates to `true`, the statement prints `2`, and `i` is incremented to `3`. And so on.
- Eventually, `i` is incremented to `10`, `10 < 10` evaluates to `false`, and the loop exits.
- Consequently, this program prints `0 1 2 3 4 5 6 7 8 9`

Iteration | `for` Loop | Off-by-one

- One of the biggest problems that new programmers have with `for` loops (and other kinds of loops) is off-by-one errors, which are logical errors (see Lecture01).
- Off-by-one errors occur when the loop iterates one too many or one too few times.
- This generally happens because the wrong relational operator is used in the conditional-expression (e.g., > instead of >=).
- *Best practice*: It is a good idea to test your loops using known values to make sure that they work as expected. A good way to do this is to test your loop with known inputs that cause it to iterate 1, 2, and 3 times. If it works for those, it will likely work for any number of iterations.

Iteration | for Loop | More Examples

- Decrement the `iteration-expression`

```
for (int i{9}; i >= 0; --i)
    std::cout << i << " ";
```

```
9 8 7 6 5 4 3 2 1 0
```

- Multiple declarations.

```
for (int i{0}, j{9}; i < 3; ++i, --j)
        std::cout << i << ' ' << j << '\n';
```

```
0 9
1 8
2 7
```

Iteration | for Loop | More Examples

- The `init-statement` can be an expression (although rarely used this way).

```
int main() {
    int n{0};

    for (std::cout << "Loop start\n";
    std::cout << "Loop test\n";
    std::cout << "Iteration " << ++n << '\n'){
        if(n > 1)
            break;//--exit the loop
    }
    std::cout << '\n';
}
```

```
Loop start
Loop test
Iteration 1
Loop test
Iteration 2
```

Iteration | for Loop | Scope

- Note: Since C++11 the scope of the variable declared in the `init-statement` is visible only in the `statement` of the loop.

```
for (int i{0}; i < 10; ++i)
    std::cout << i << "\n";

std::cout << i; //--error: 'i' was not declared in this scope
```

- Note: Prior to C++11 this variable could be accessed outside the loop body. That is, the error we have in the code above was not an error in C++ versions earlier to C++11. You may find this kind of code if you use old C++ code.

Iteration | Range-based `for` Loop

- Range-based `for` loop syntax (since C++11) is a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container.

```
for (range-declaration: range-expression)
    statement
```

- `range-declaration` is a declaration of a named variable, whose type is the type of the element of the sequence represented by `range-expression`, or a reference to that type. Often uses the `auto` specifier for automatic type deduction.
- `range-expression`: Any expression that represents a suitable sequence.
- `statement`: Any statement, usually a compound statement, which is the body of the loop.

Iteration | Range-based `for` Loop

```
//--Sum all the value of each element of a vector
std::vector<double> scores{5.0, 13.0, 3.5};//--3 elements in the vector
double sum{};//--Zero initialization

for (double score: scores)//--This will iterate 3 times
    sum += score;//-- sum = sum + score

std::cout << "Total score: " << sum << std::endl;
```

- Every iteration of the loop creates a local variable `score` and initializes it to the next element of `scores`.
  - 1st iteration: `score=5.0` and `sum = 0 + 5.0 = 5.0`
  - 2nd iteration: `score=13.0` and `sum = 5.0 + 13.0 = 18.0`
  - 3rd iteration: `score=3.5` and `sum = 18.0 + 3.5 = 21.5`

Iteration | Range-based `for` Loop | The `auto` Specifier

- The `auto` specifier specifies that the type of the variable that is being declared will be automatically deduced from its initializer.

```
std::vector<double> scores {5.0, 13.0, 3.5};
double sum{0};

for (auto score: scores)//--Using auto specifier
    sum += score;

std::cout << "Total score: " << sum << std::endl;
```

- With the `auto` specifier, the type of the variable `score` is deduced to be a `double`.

Iteration | Range-based `for` Loop | The `auto` Specifier

- If you are working with C++11, then you can use the `auto` specifier to deduce the type of regular variables.
- If you are working with C++17, you go even further by omitting the type of the data type when working with `std::vector`.
- Compare the code below with the code in the previous slide.

```
std::vector scores {5.0, 13.0, 3.5};//--Removed double
auto sum{0.0};//--0.0 is used by the compiler to deduce the type of sum

for (auto score: scores)
    sum += score;

std::cout << "Total score: " << sum << std::endl;
```

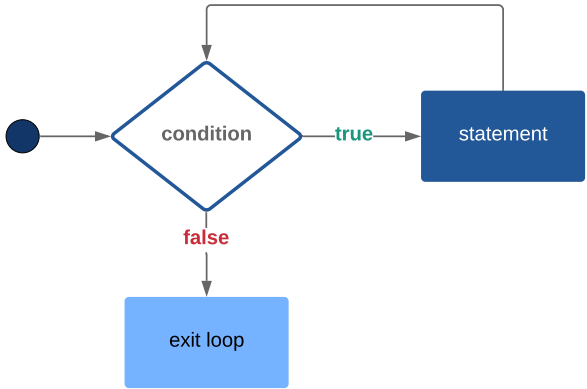
Iteration | `while` Loop

- Executes a statement repeatedly, until the value of condition becomes `false`. The test takes place before each iteration.

```
while (condition)
    statement
```

- `condition`: Any expression which is contextually convertible to `bool` or a declaration of a single variable with a brace-or-equals initializer. This expression is evaluated before each iteration, and if it yields `false`, the loop is exited.
- `statement`: Any statement, typically a compound statement, which is the body of the loop.

Iteration | `while` Loop



Iteration | while Loop | Examples

```
int i{};
while (i<=5){
    if (i%2 == 0)
        std::cout << i << " is an even number\n";
    i++;
}
```

```
0 is an even number
2 is an even number
4 is an even number
```

Iteration | while Loop | Examples

- Below is a typical use of while loops. The program will loop until the user enters a number which evaluates the while loop condition to true.

```
int number{};
std::cout << "Enter an integer number between 0 and 10: ";
std::cin >> number;

while (number > 10 || number < 0){
    std::cout << "Out of range\n";
    std::cout << "Enter an integer number between 0 and 10: ";
    std::cin >> number;
    }
std::cout << "The number you entered is: " << number << "\n";
```

Iteration | `while` Loop | Examples

- Another way of performing the previous example using a `while` loop and an `if-else` statement.

```
bool done {false};
int number{};

while (!done){
    cout << "Enter an integer number between 0 and 10: ";
    cin >> number;
    if (number > 10  number < 0)
        cout << "Out of range\n";
    else{
        cout << "The number you entered is: " << number << endl;
        done = true;
    }
}
```

Iteration | `while` Loop | `for` Vs. `while` Loops

- `for` loops and while loops can both be used to iterate until a condition is satisfied.
- Prefer the use of `for` loops until a condition is satisfied and when you know how many times to iterate.

```
for (int i{0};i<10;++i){
    /*iterate until i>=10*/
}
```

- Prefer the use of `while` loops until a condition is satisfied and when it is unsure how many times you need to iterate.

```
int i{};
while (i<10){
    /*code that will increment the value of i*/
    /*we do not know how many times it will iterate*/
}
```

Iteration | `while` Loop | `for` Vs. `while` Loops

```
bool flag{false};
int i{}, j{}, val{};

while (!flag){
    val += i*2 + j*3;
    if (val > 100)
        flag=true;

    ++i;
    ++j;
}

    std::cout << val << std::endl;
```

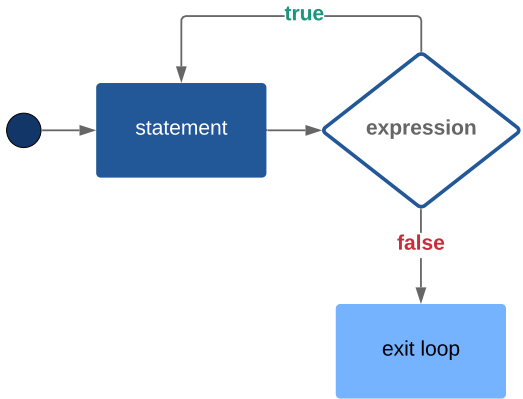
- I don't know how many iterations it will take for `val > 100` (which exits the loop), therefore a `while` loop is more appropriate in this case.

Iteration | `do-while` Loop

```
do {statement} while (expression);
```

- Executes a `statement` repeatedly, until the value of `expression` becomes `false`. The test takes place after each iteration.
- `statement` is always executed at least once, even if `expression` always yields `false`. If it should not execute in this case, a `while` or `for` loop may be more appropriate.
- If the execution of the loop needs to be terminated at some point, a `break;` statement can be used.
- If the execution of the loop needs to be continued at the end of the loop body, a `continue;` statement can be used.

Iteration | do-while Loop



Iteration | do-while Loop

- The first iteration of the code asks the user to enter 3 numbers and then multiplies those numbers.
- The second iteration of the code depends on the user's choice to do another multiplication or not.

```
//--Multiply 3 numbers
char selection{};
do{
    int num1{}, num2{}, num3{};

    std::cout << "enter 3 integer numbers separated by a space: ";
    std::cin >> num1 >> num2 >> num3;

    int result{num1 * num2 * num3};
    std::cout << "result is " << result << std::endl;

    std::cout << "Do another multiplication? (Y/N)" << std::endl;
    std::cin >> selection;
} while (selection == 'Y'  selection == 'y');
```

Iteration | do-while Loop

- The `continue` keyword forces the next iteration of the loop to take place, skipping any code in between.

```
int a{};
do{
    if(a == 2) {
        a++;//-- increment a
        continue;//--skip everything else in the body of the loop
    }
    std::cout << "Value of a: " << a << std::endl;
    a++;
} while (a < 5);
```

```
Value of a: 0
Value of a: 1
Value of a: 3
Value of a: 4
```

Iteration | do-while Loop

- A `break` statement inside a loop forces the loop to terminate immediately and program control resumes at the next statement following the loop.

```
int a{};
do{
    if(a == 2) {
        //-- terminate the loop
        break;
    }
    std::cout << "Value of a: " << a << std::endl;
    a++;
} while (a < 5);
```

```
Value of a: 0
Value of a: 1
```

Iteration | Nested Loops

- Loop nested inside another loop.
- Can be as many levels deep as you need to.
- Very useful for multidimensional data structures.
- We have an outer and an inner loop in nested loops.

Iteration | Nested Loops

```
//--2D vector: vector containing vectors
std::vector<std::vector<int>> vector_2d{
    {1,2,3},
    {4,5,6,7},
    {8,9,10,11,12}
};

for (auto vector_1d: vector_2d){//--Get each vector in vector_2d
    for (auto val:vector_1d){//--Get each element (int) of vector_1d
        std::cout << val << " ";
    }
    std::cout << std::endl;//--New line
}
```

Iteration | Exercise #4

- Write the pseudo code for the problem below.
- Declare and initialize a `vector<int> vec;`
- Find the sum of the product of all pairs of vector elements.
- Examples:
  - Given the vector `vec={2,4,5}`, the possible pairs are `(2,4), (2,5), and (4,5)`. The result is `(2*4)+(2*5)+(4*5)=38`.

    ```
    Result is 38
    ```

  - Given the vector `vec={1,2,3,4}`, the result is `(1*2)+(1*3)+(1*4)+(2*3)+(2*4)+(3*4)=35`.

    ```
    Result is 35
    ```

- If the vector is empty, display `Vector is empty`
- If the vector has only 1 element, then return this element.