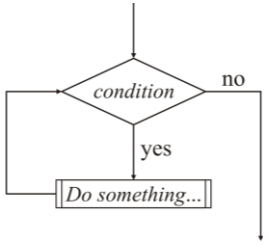
Outline

- In this lesson, we will:
  - Describe do-while loops
    - A variation of while loops
  - Look at two applications:
    - Required input from users or sensors
    - Fixed-point iteration
  - We will look at an engineering principle of static determination

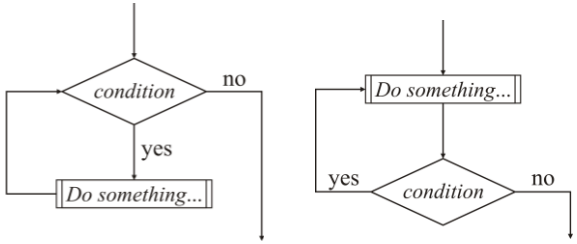
Looping statements

- A while loop has the following flow chart
  - It requires that the condition is initially true



Looping statements

- In some circumstances, it may not be possible to determine if any condition is true prior to any execution



Do-while loops

- A do-while loop is essentially a while loop that does not check the condition before the first time the block of statements is executed

```
do {
    The looped block of statements
     - to be executed as long as the
       condition is 'true'
} while ( Boolean-valued condition );

// Continue executing here as soon as the
// condition evaluates to 'false'
```

Note the semicolon

Example: user input

- For example, suppose you absolutely require input from the user
  - This could be checked before, but leads to repeated code

```
void high_low_game() {
    int target{(std::rand() % 100) + 1};
    int guess{};
    std::cout << "Pick a number between 1 and 100: ";
    std::cin >> guess;

    while ( guess != target ) {
        if ( guess < target ) {
            std::cout << "Low" << std::endl;
        } else {
            assert( guess > target );
            std::cout << "High" << std::endl;
        }

        std::cout << "Pick a number between 1 and 100: ";
        std::cin >> guess;
    }

    std::cout << "Good guess" << std::endl;
}
```


Example: user input

- The repetition of the code can be avoided

```
void high_low_game() {
    int target{(std::rand() % 100) + 1};
    int guess{};

    do {
        std::cout << "Pick a number between 1 and 100: ";
        std::cin >> guess;

        if ( guess < target ) {
            std::cout << "Low" << std::endl;
        } else if ( guess > target ) {
            std::cout << "High" << std::endl;
        }
    } while ( guess != target );

    std::cout << "Good guess" << std::endl;
}
```

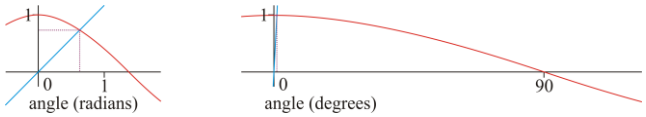

Example: user input

- Try this yourself:

```
#include <iostream>
#include <cstdlib>

// Function declarations
void high_low_game();
int main();

// Function definitions
int main() {
    high_low_game();
    return 0;
}

void high_low_game() {
    int target{(std::rand() % 100) + 1};
    int guess{};

    do {
        std::cout << "Pick a number between 1 and 100: ";
        std::cin >> guess;

        if ( guess < target ) {
            std::cout << "Low" << std::endl;
        } else if ( guess > target ) {
            std::cout << "High" << std::endl;
        }
    } while ( guess != target );

    std::cout << "Good guess" << std::endl;
}
```

Fixed-point iteration

- As a second example: significant scientific computation starts with an approximation and iterates to find a better solution
  - If successive iterations are sufficiently close, we assume that it is a good approximation
  - This requires at least one iteration to see if we are sufficiently close
  - For example, to find a point x such that $x = \cos(x)$, under some circumstances, all you must do is pick a starting point and repeatedly apply cos to an initial value



Fixed-point iteration

- You can do this with your calculator:

Using degrees:

```
1.0
0.9998476951563912
0.9998477415452118
0.9998477415310838
0.9998477415310881
0.9998477415310881
```

Using radians:

```
1.0
0.5403023058681397
0.8575532158463934
0.6542897904977792
0.7934803587425656
0.7013687736227565
0.7639596829006542
0.7221024250267078
```

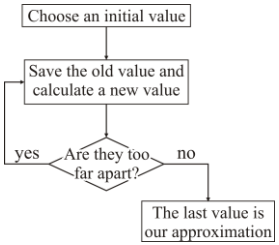
Ultimately, it converges to:

```
0.7390851332151606
```

  - You keep iterating until two consecutive solutions are *close enough*

Fixed-point iteration

- A simplified flow chart is as follows:



Fixed-point iteration

- Normally, however, the condition is affected by the action of the looped block of statements

```
#include <cmath>
double cosine_fixed_point();

double cosine_fixed_point() {
    double x_current{1.0};
    double x_previous;

    do {
        x_previous = x_current;

        x_current = std::cos( x_current );
    } while ( std::abs( x_previous - x_current ) > 1e-10 );

    return x_current;
}
```

Fixed-point iteration

- Suppose you try finding the fixed point of cot(x)
  - There are infinitely many fixed points:

    ±0.8603335890193798
    ±3.425618459481728
    ±6.437298179171947
    ±9.529334405361964

  - The code requires hardly any modification…

```
do {
    x_previous = x_current;

    x_current = 1.0/std::tan( x_current );
} while ( std::abs( x_previous - x_current ) > 1e-10 );
```


Fixed-point iteration

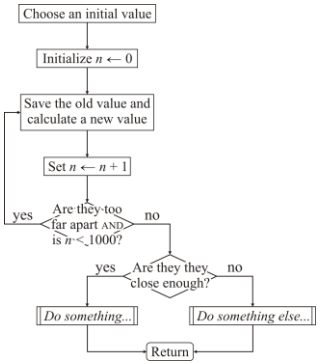
- Problem: it runs forever
  - In this case, it is in a non-terminating loop
  - How can we deal with this?
- In engineering, it is essential to ensure that no loop runs forever, even accidentally, as from the JPL coding standard:

  Rule 3 (loop bounds)

  All loops ***shall*** have a statically determinable upper-bound on the maximum number of loop iterations.
- Here, *static* means that a reader can look at the code and determine the upper bound **before** the source code is compiled

Fixed-point iteration

- Here we stop if we exceed 1000 iterations:



Fixed-point iteration

- Here is our implementation:

```
double fixed_point();

void fixed_point() {
    double x_current{1.0};
    double x_previous{};
    unsigned int num_iterations{0};

    do {
        ++num_iterations;
        x_previous = x_current;
        x_current = std::cos( x_current );
    } while ( (std::abs( x_previous - x_current ) > 1e-10)
              && (num_iterations < 1000) );

    if ( std::abs( x_previous - x_current ) <= 1e-10 ) {
        std::cout << "Fixed point: " << x_current << std::endl;
    } else {
        std::cout << "Did not converge..." << std::endl;
    }
}
```

Fixed-point iteration

- Try this yourself:

```
#include <cmath>
#include <iostream>

// Function declarations
void fixed_point();
int main();

// Function definitions
int main() {
    fixed_point();
    return 0;
}

void fixed_point() {
    double x_current{1.0};
    double x_previous{};
    unsigned int num_iterations{0};

    do {
        ++num_iterations;
        x_previous = x_current;
        x_current = std::cos( x_current );
    } while ( (std::abs( x_previous - x_current ) > 1e-10)
            && (num_iterations < 1000) );

     if ( std::abs( x_previous - x_current ) <= 1e-10 ) {
         std::cout << "Fixed point: " << x_current << std::endl;
     } else {
         std::cout << "Did not converge..." << std::endl;
     }
}
```


Summary

- Following this lesson, you now
  - Understand how to implement do-while loops in C++
  - Have seen how to repeatedly respond to data from the user or sensors
  - Have had exposure to numerical algorithms
  - Understand why it is necessary to limit the number of times the statement block is executed

References

[1] Wikipedia
https://en.wikipedia.org/wiki/Do_while_loop

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.



Disclaimer

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.