Exercise Session — Computer Science — 03

Expressions, Loops, Calculating Sums, Scopes

Overview

Today's Plan

Feedback regarding **code** expert
Expressions
Loops
Calculating Sums
Scopes

1. Feedback regarding **code** expert

Better explanation for short-circuits

- Short circuit rule **actually** complies with the precedence ranking of operations.

Better explanation for short-circuits

- Short circuit rule **actually** complies with the precedence ranking of operations.
- **Operator precedence** in C++ is primarily used to determine the **grouping of operators and operands**, essentially deciding where parentheses would be if they were added.

Better explanation for short-circuits

- Short circuit rule **actually** complies with the precedence ranking of operations.
- **Operator precedence** in C++ is primarily used to determine the **grouping of operators and operands**, essentially deciding where parentheses would be if they were added.
- **Operator precedence** decides in which order the operators are **applied**, but **not** how they are **executed**.

Better explanation for short-circuits

- Short circuit rule **actually** complies with the precedence ranking of operations.
- **Operator precedence** in C++ is primarily used to determine the **grouping of operators and operands**, essentially deciding where parentheses would be if they were added.
- **Operator precedence** decides in which order the operators are **applied**, but **not** how they are **executed**.
- The C++ compiler has some freedom in deciding the order of evaluation of subexpressions, as long as the final result remains consistent with the grouping.

Better explanation for short-circuits

- Short circuit rule **actually** complies with the precedence ranking of operations.
- **Operator precedence** in C++ is primarily used to determine the **grouping of operators and operands**, essentially deciding where parentheses would be if they were added.
- **Operator precedence** decides in which order the operators are **applied**, but **not** how they are **executed**.
- The C++ compiler has some freedom in deciding the order of evaluation of subexpressions, as long as the final result remains consistent with the grouping.
- C++ will use different optimizations to speed up the execution, such as **short circuits**.

Better explanation for short-circuits

- Short circuit rule **actually** complies with the precedence ranking of operations.
- **Operator precedence** in C++ is primarily used to determine the **grouping of operators and operands**, essentially deciding where parentheses would be if they were added.
- **Operator precedence** decides in which order the operators are **applied**, but **not** how they are **executed**.
- The C++ compiler has some freedom in deciding the order of evaluation of subexpressions, as long as the final result remains consistent with the grouping.
- C++ will use different optimizations to speed up the execution, such as **short circuits**.

Everything you should need about precedence

- **L-Values** have a memory address (e.g., int a) and values can be assigned to them
- **R-Values** have no memory address

Prec.	*Operator*	*Input*	*Output*	*Associativ.*
2	`a++`, `a--`	L-Value	R-Value	Right
3	`++a`, `--a`	L-Value	L-Value	Right
3	`*a`, `&a`	L/R-Value	L-Value	Right
5	`*`, `/`, `%`	R-Value	R-Value	Left
6	`+`, `-`	R-Value	R- Value	Left
9	`>`, `>=`, `<`, `<=`	R-Values	R- Value	Left
10	`==`, `!=`	R- Values	R- Value	Left
	`>>`	L- Values	L- Value	Left
	`<<`	L/R Values	L- Value	Left
14	`&&`	R- Values	R- Value	Left
15	`||`	R- Values	R- Value	Left
16	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	L/R Values	L- Value	Right

- One more: `!x` = L-value

General things regarding code expert

- It is advised to use `int` instead of `unsigned int` in this course for safety.
- Arithmetic with `unsigned int` can lead to silent wrapping (overflow or underflow) when the value goes below 0 or exceeds the maximum value.
- Mixing signed (`int`) and unsigned (`unsigned int`) types in comparisons or arithmetic can lead to confusing or incorrect results.

Any questions regarding **code** expert on your part?

2. Expressions

Types

Types covered so far

Types

Types covered so far

- logic variables: `bool` {`false`, `true`}

Types

Types covered so far

- logic variables: `bool` {`false`, `true`}
- integers: `unsigned int`, `int` {`-7, 2, 0`}

Types

Types covered so far

- logic variables: `bool` {`false`, `true`}
- integers: `unsigned int`, `int` {`-7, 2, 0`}
- floating point numbers: `float`, `double` {`1.4, -4.3, 7.0`}

Types

Types covered so far

- logic variables: `bool` {`false`, `true`}
- integers: `unsigned int`, `int` {`-7, 2, 0`}
- floating point numbers: `float`, `double` {`1.4, -4.3, 7.0`}

Sometimes, multiple types are present in the same expression.
How do different types interact?

Types

Types covered so far

- logic variables: `bool` {`false`, `true`}
- integers: `unsigned int`, `int` {`-7, 2, 0`}
- floating point numbers: `float`, `double` {`1.4, -4.3, 7.0`}

Sometimes, multiple types are present in the same expression.
How do different types interact?

Generality order of types

Types

Types covered so far

- logic variables: `bool` {`false`, `true`}
- integers: `unsigned int`, `int` {`-7, 2, 0`}
- floating point numbers: `float`, `double` {`1.4, -4.3, 7.0`}

Sometimes, multiple types are present in the same expression.
How do different types interact?

Generality order of types

```
bool <
```

Types

Types covered so far

- logic variables: `bool` {`false`, `true`}
- integers: `unsigned int`, `int` {`-7, 2, 0`}
- floating point numbers: `float`, `double` {`1.4, -4.3, 7.0`}

Sometimes, multiple types are present in the same expression.
How do different types interact?

Generality order of types

```
bool < int < unsigned int <
```

Types

Types covered so far

- logic variables: `bool` {`false`, `true`}
- integers: `unsigned int`, `int` {`-7, 2, 0`}
- floating point numbers: `float`, `double` {`1.4, -4.3, 7.0`}

Sometimes, multiple types are present in the same expression.
How do different types interact?

Generality order of types

```
bool < int < unsigned int < float < double
```

Types always convert to the more general type in an expression

Mental model of types

Type (literal)	Approximates

Mental model of types

Type (literal)	Approximates
`bool`	{`false`, `true`}

Mental model of types

Type (literal)	Approximates
`bool`	{`false`, `true`}
`unsigned int` (u)	$\mathbb{N}$

Mental model of types

Type (literal)	**Approximates**
`bool`	{`false`, `true`}
`unsigned int` (u)	$\mathbb{N}$
`int`	$\mathbb{Z}$

Mental model of types

Type (literal)	**Approximates**
`bool`	$\{$`false`, `true`$\}$
`unsigned int` (u)	$\mathbb{N}$
`int`	$\mathbb{Z}$
`float` (f)	$\mathbb{R}$

Mental model of types

Type (literal)	Approximates
`bool`	$\{$`false`, `true`$\}$
`unsigned int` (u)	$\mathbb{N}$
`int`	$\mathbb{Z}$
`float` (f)	$\mathbb{R}$
`double`	$\mathbb{R}$, but *double* precision

Evaluating Types I

```
std::cout << 5.0/2 << std::endl;
// what type and value will this return and why?
```

Evaluating Types I

```
std::cout << 5.0/2 << std::endl;
// what type and value will this return and why?
```

Solution

`double`, `2.5`, since the `int 2` gets turned into a `double 2.0` first in order to calculate this expression.

Evaluating Types II

```
std::cout << (1/2)*5.0/2 << std::endl;
// what type and value will this return and why?
```

Evaluating Types II

```
std::cout << (1/2)*5.0/2 << std::endl;
// what type and value will this return and why?
```

Solution

`double`, 0 because the left expression 1/2 gets evaluated first, which evaluates to 0, since it's an integer division. The rest is trivial, since 0*anything evaluates to 0. That 0 will be of type `double`.

Literals

Literals

There are certain letters which are assigned certain meanings regarding types. If you want to tell the compiler *"Hey, don't treat this `2.0` as a **`double`**, but instead as a **`float`**"* you have to put an `f` at the end of the value. Like this:

Literals

There are certain letters which are assigned certain meanings regarding types. If you want to tell the compiler *"Hey, don't treat this `2.0` as a **`double`**, but instead as a **`float`*"* you have to put an `f` at the end of the value. Like this:

```
std::cout << (5/2)*5.0f/2 << std::endl;
```

Evaluating Types III

```
std::cout << (5/2)*5.0f/2 << std::endl;
// what type and value will this return and why?
```

Evaluating Types III

```
std::cout << (5/2)*5.0f/2 << std::endl;
// what type and value will this return and why?
```

Solution

`float`, `5.0`, can be written as `5.0f`.

First, the `5/2` gets evaluted which results in `2` (integer division). Then `2.0f*5.0f`: The `int 2` became a `float` because that is the more general type (in this expression). Ditto for `/2` later.

Exercise I

1. Which of the following character sequences are not C++ expressions, and why not? Here, `x` and `y` are variables of type `int`.
   a) `(y++ < 0 && y < 0) + 2.0`
   b) `y = (x++ = 3)`
   c) `3.0 + 3 - 4 + 5`
   d) `5 % 4 * 3.0 + true * x++`
2. For all of the valid expressions that you have identified above, decide whether these are l-values or r-values and explain your decision.
3. Determine the values of the expressions and explain how these values are obtained. Assume that initially `x == 1` and `y == -1`.

Expression Evaluation - Solutions a)

```
(y++ < 0 && y < 0) + 2.0
```

Expression Evaluation - Solutions a)

```
(y++ < 0 && y < 0) + 2.0

(-1 < 0 && y < 0) + 2.0  // after this step: y==0
```

Expression Evaluation - Solutions a)

```
(y++ < 0 && y < 0) + 2.0

(-1 < 0 && y < 0) + 2.0  // after this step: y==0
(true && y < 0) + 2.0
```

Expression Evaluation - Solutions a)

```
(y++ < 0 && y < 0) + 2.0

(-1 < 0 && y < 0) + 2.0  // after this step: y==0
(true && y < 0) + 2.0
(true && false) + 2.0
```

Expression Evaluation - Solutions a)

```
(y++ < 0 && y < 0) + 2.0

(-1 < 0 && y < 0) + 2.0  // after this step: y==0
(true && y < 0) + 2.0
(true && false) + 2.0
(false) + 2.0
```

Expression Evaluation - Solutions a)

```
(y++ < 0 && y < 0) + 2.0

(-1 < 0 && y < 0) + 2.0  // after this step: y==0
(true && y < 0) + 2.0
(true && false) + 2.0
(false) + 2.0
0.0 + 2.0
```

Expression Evaluation - Solutions a)

```
(y++ < 0 && y < 0) + 2.0

(-1 < 0 && y < 0) + 2.0  // after this step: y==0
(true && y < 0) + 2.0
(true && false) + 2.0
(false) + 2.0
0.0 + 2.0
2.0
```

Expression Evaluation - Solutions a)

```
(y++ < 0 && y < 0) + 2.0

(-1 < 0 && y < 0) + 2.0  // after this step: y==0
(true && y < 0) + 2.0
(true && false) + 2.0
(false) + 2.0
0.0 + 2.0
2.0
```

r-Value

Expression Evaluation - Solutions b)

```
y = (x++ = 3)
```

Expression Evaluation - Solutions b)

```
y = (x++ = 3)
```

Invalid

Expression Evaluation - Solutions c)

```
3.0 + 3 - 4 + 5
```

Expression Evaluation - Solutions c)

```
3.0 + 3 - 4 + 5

((3.0 + 3) - 4) + 5
```

Expression Evaluation - Solutions c)

```
3.0 + 3 - 4 + 5
```

```
((3.0 + 3) - 4) + 5
((3.0 + 3.0) - 4) + 5
```

Expression Evaluation - Solutions c)

```
3.0 + 3 - 4 + 5

((3.0 + 3) - 4) + 5
((3.0 + 3.0) - 4) + 5
(6.0 - 4) + 5
```

Expression Evaluation - Solutions c)

```
3.0 + 3 - 4 + 5

((3.0 + 3) - 4) + 5
((3.0 + 3.0) - 4) + 5
(6.0 - 4) + 5
(6.0 - 4.0) + 5
```

Expression Evaluation - Solutions c)

```
3.0 + 3 - 4 + 5

((3.0 + 3) - 4) + 5
((3.0 + 3.0) - 4) + 5
(6.0 - 4) + 5
(6.0 - 4.0) + 5
2.0 + 5
```

Expression Evaluation - Solutions c)

```
3.0 + 3 - 4 + 5

((3.0 + 3) - 4) + 5
((3.0 + 3.0) - 4) + 5
(6.0 - 4) + 5
(6.0 - 4.0) + 5
2.0 + 5
2.0 + 5.0
```

Expression Evaluation - Solutions c)

```
3.0 + 3 - 4 + 5

((3.0 + 3) - 4) + 5
((3.0 + 3.0) - 4) + 5
(6.0 - 4) + 5
(6.0 - 4.0) + 5
2.0 + 5
2.0 + 5.0
7.0
```

Expression Evaluation - Solutions c)

```
3.0 + 3 - 4 + 5

((3.0 + 3) - 4) + 5
((3.0 + 3.0) - 4) + 5
(6.0 - 4) + 5
(6.0 - 4.0) + 5
2.0 + 5
2.0 + 5.0
7.0
```

r-Value

Expression Evaluation - Solutions d)

```
5 % 4 * 3.0 + true * x++
```

Expression Evaluation - Solutions d)

```
5 % 4 * 3.0 + true * x++
```

```
((5 % 4) * 3.0) + (true * (x++))
```

Expression Evaluation - Solutions d)

```
5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))
(1 * 3.0) + (true * (x++))
```

Expression Evaluation - Solutions d)

```
5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))
(1 * 3.0) + (true * (x++))
(1.0 * 3.0) + (true * (x++))
```

Expression Evaluation - Solutions d)

```
5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))
(1 * 3.0) + (true * (x++))
(1.0 * 3.0) + (true * (x++))
3.0 + (true * (x++))
```

Expression Evaluation - Solutions d)

```
5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))
(1 * 3.0) + (true * (x++))
(1.0 * 3.0) + (true * (x++))
3.0 + (true * (x++))
3.0 + (true * 1)
```

Expression Evaluation - Solutions d)

```
5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))
(1 * 3.0) + (true * (x++))
(1.0 * 3.0) + (true * (x++))
3.0 + (true * (x++))
3.0 + (true * 1)
3.0 + (1 * 1)
```

Expression Evaluation - Solutions d)

```
5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))
(1 * 3.0) + (true * (x++))
(1.0 * 3.0) + (true * (x++))
3.0 + (true * (x++))
3.0 + (true * 1)
3.0 + (1 * 1)
3.0 + 1
```

Expression Evaluation - Solutions d)

```
5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))
(1 * 3.0) + (true * (x++))
(1.0 * 3.0) + (true * (x++))
3.0 + (true * (x++))
3.0 + (true * 1)
3.0 + (1 * 1)
3.0 + 1
3.0 + 1.0
```

Expression Evaluation - Solutions d)

```
5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))
(1 * 3.0) + (true * (x++))
(1.0 * 3.0) + (true * (x++))
3.0 + (true * (x++))
3.0 + (true * 1)
3.0 + (1 * 1)
3.0 + 1
3.0 + 1.0
4.0
```

Expression Evaluation - Solutions d)

```
5 % 4 * 3.0 + true * x++

((5 % 4) * 3.0) + (true * (x++))
(1 * 3.0) + (true * (x++))
(1.0 * 3.0) + (true * (x++))
3.0 + (true * (x++))
3.0 + (true * 1)
3.0 + (1 * 1)
3.0 + 1
3.0 + 1.0
4.0
```

r-Value

3. Loops

Loop Correctness

Can a user of the program observe the difference between the output produced by these three loops? If yes, how? Assume that `n` is a variable of type `unsigned int` whose value is given by the user.

Loop Correctness

Can a user of the program observe the difference between the output produced by these three loops? If yes, how? Assume that `n` is a variable of type `unsigned int` whose value is given by the user.

```
//////////////////////////
unsigned int n;
std::cin >> n;
unsigned int i;

// loop 1 ////////////////
for (i = 1; i <= n; ++i) {
  std::cout << i << "\n";
}
```

```
// loop 2 ////////////////
i = 0;
while (i < n) {
    std::cout << ++i << "\n";
}
// loop 3 ////////////////
i = 1;
do {
  std::cout << i++ << "\n";
} while (i <= n);
```

Loop Correctness - Solution

Solution

There are the following differences:

- Unlike loops 1 and 2, loop 3 does output `1` for input `n == 0` because the statement in a `do`-loop is always executed once before the condition is checked
- If n is the largest possible integer, then the loops 1 and 3 may be infinite because the condition `i <= n` is going to be true for all possible `i`

Questions?

for → while

```
// TASK: Convert the following
// for-loop into an
// equivalent while-loop:

for (int i = 0; i < n; ++i) {
    // BODY
}
```

for → while

```cpp
// TASK: Convert the following
// for-loop into an
// equivalent while-loop:

for (int i = 0; i < n; ++i) {
    // BODY
}
```

```cpp
// SOLUTION

int i = 0;

while(i < n){
    // BODY
    ++i;
}
```

while → for

```
// TASK: Convert the following
// while-loop into an
// equivalent for-loop:

while(condition){
    // BODY
}
```

while → for

```
// TASK: Convert the following
// while-loop into an
// equivalent for-loop:

while(condition){
    // BODY
}
```

```
// SOLUTION

for(;condition;){
    // BODY
}
```

do-while → for

```
// TASK: Convert the following
// do-while-loop into an
// equivalent for-loop:

do{
    // BODY
}while(condition)
```

do-while → for

```
// TASK: Convert the following
// do-while-loop into an
// equivalent for-loop:

do{
    // BODY
}while(condition)
```

```
// SOLUTION

// BODY
for(;condition;){
    // BODY
}
```

Questions?

4. Calculating Sums

From Series to Loop

Mathematical sums can be turned into loops

$$\sum_{i=0}^{n} f(i)$$

From Series to Loop

Mathematical sums can be turned into loops

$$\sum_{i=0}^{n} f(i)$$

Becomes

```
int n = 0;
int sum = 0;

for(int i = 0; i <= n; i++){
  sum += f(i);
}
```

Warmup Exercise

Consider the formula

$$\frac{1}{n!}$$

Warmup Exercise

Consider the formula

$$\frac{1}{n!} = \frac{1}{1} \cdot \frac{1}{2} \cdot \dots \cdot \frac{1}{n}$$

Warmup Exercise

Consider the formula

$$\frac{1}{n!} = \frac{1}{1} \cdot \frac{1}{2} \cdot \dots \cdot \frac{1}{n}$$

How could one implement this as a ("multiplicative") series?

$$\frac{1}{n!}$$

Warmup Exercise

Consider the formula

$$\frac{1}{n!} = \frac{1}{1} \cdot \frac{1}{2} \cdot \dots \cdot \frac{1}{n}$$

How could one implement this as a ("multiplicative") series?

$$\frac{1}{n!} = \prod_{i=1}^{n} \frac{1}{i}$$

How do we turn this piece of math into a piece of C++ code?

Warmup Exercise

Consider the formula

$$\frac{1}{n!} = \frac{1}{1} \cdot \frac{1}{2} \cdot \dots \cdot \frac{1}{n}$$

How could one implement this as a ("multiplicative") series?

$$\frac{1}{n!} = \prod_{i=1}^{n} \frac{1}{i}$$

How do we turn this piece of math into a piece of C++ code?

```
int main(){

    int n;           // user input
    double result;   // main output






    std::cout << result
              << std::endl;

    return 0;
}
```

Warmup Exercise - Example Solution

```
int main(){
    int n;
    double result = 1;
    int i = 1;

    std::cin >> n;

    while(i <= n){
        result = result/i;
        i++;
    }

    std::cout << result << std::endl;

    return 0;
}
```

From Series to Loop

Taylor Series on code expert

Write a program that calculates $\sin(x)$ up to six decimal places

Hint: Use the MacLaurin Series. Hint: How would you compute (n + 1)-th term when you have the n-th term?

Hint: What loop should be used here?

Hint: Try to solve the exercise without considering the precision at first.

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

From Series to Loop

Taylor Series on code expert

Write a program that calculates $\sin(x)$ up to six decimal places
Hint: Use the MacLaurin Series. Hint: How would you compute (n + 1)-th term when you have the n-th term?
Hint: What loop should be used here?
Hint: Try to solve the exercise without considering the precision at first.

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

Task

- Try with pen and paper

From Series to Loop

Taylor Series on code expert

Write a program that calculates $\sin(x)$ up to six decimal places
Hint: Use the MacLaurin Series. Hint: How would you compute (n + 1)-th term when you have the n-th term?
Hint: What loop should be used here?
Hint: Try to solve the exercise without considering the precision at first.

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

Task

- Try with pen and paper
- Try implementing it together with person next to you in **code** expert

From Series to Loop - Solution

From Series to Loop - Solution

```cpp
#include <iostream>

int main () {

  double x;
  std::cin >> x;

  double numtor = x;
  double denomtor = 1;

  double sum = x;
  double term;
  double term_abs;
  int n = 1;
```

```cpp
  do {
      numtor *= -(x * x);
      denomtor *= (2 * n) * (2 * n + 1);
      term = numtor / denomtor;
      sum += term;
      if (term < 0) {
        term_abs = -term;
      } else {
        term_abs = term;
      }
      ++n;
  } while (term_abs > 0.000001);

  std::cout << sum << std::endl;
  return 0;
}
```

Questions?

5. Scopes

Scopes

Question

In this week’s lecture, a new concept was introduced called "variable scopes". Does anyone remember what variable scopes are and why do we need them?

Scopes

Question

In this week's lecture, a new concept was introduced called "variable scopes". Does anyone remember what variable scopes are and why do we need them?

Answer

Scopes define the code segments of our program in which a variable (l-value) exists. The scope of a variable starts at the point of its definition and ends at the end of the block where it was defined. For example:

```
if (x < 7){
    int a = 8;              // <-- a's variable scope BEGINS here!
    std::cout << a;         // Fine, prints 8.
}                           // <-- a's variable scope ENDS here!
std::cout << a;             // Compiler error, a does not exist.
```

Bug?

A way to supposedly fix the compilation error would be this:

```
int a = 2;

if (x < 7) {
    int a = 8;
    std::cout << a;
}

std::cout << a;
```

Question

What this program is going to print if `x==2`?

Bug?

A way to supposedly fix the compilation error would be this:

```
int a = 2;

if (x < 7) {
    int a = 8;
    std::cout << a;
}

std::cout << a;
```

Question

What this program is going to print if `x==2`?

Answer

It's going to print 82.

Bug?

A way to supposedly fix the compilation error would be this:

```
int a = 2;

if (x < 7) {
    int a = 8;
    std::cout << a;
}

std::cout << a;
```

Question

What this program is going to print if `x==2`?

Answer

It's going to print 82.

Why? See Program Tracing Guide

Bug?

Question

What is the scopes of `sum`, `i`, and `a` in the following example?

```
int sum = 0;

for (int i = 0; i < 5; ++i) {
    int a;
    std::cin >> a;
    sum += a;
}
```

Answer

Bug?

Question

What is the scopes of `sum`, `i`, and `a` in the following example?

```
int sum = 0;

for (int i = 0; i < 5; ++i) {
    int a;
    std::cin >> a;
    sum += a;
}
```

Answer

`sum` (At least) the entire snippet

Bug?

Question

What is the scopes of `sum`, `i`, and `a` in the following example?

```
int sum = 0;

for (int i = 0; i < 5; ++i) {
    int a;
    std::cin >> a;
    sum += a;
}
```

Answer

`sum` (At least) the entire snippet

`i` The entire `for`-loop

Bug?

Question

What is the scopes of `sum`, `i`, and `a` in the following example?

```
int sum = 0;

for (int i = 0; i < 5; ++i) {
    int a;
    std::cin >> a;
    sum += a;
}
```

Answer

`sum` (At least) the entire snippet

`i` The entire `for`-loop

`a` One loop iteration. In other words, at the beginning of the loop body `a` is *not* guaranteed to have the value it had at the end of the loop body in the previous loop iteration.

Questions?

6. Outro

General Questions?

See you next time

Have a nice week!