# 6. Functions

Ján Dugáček

October 25, 2018

# Table of Contents

# Motivation

- Can you think of a comfortable way to calculate $A \cdot (B \cdot A) \cdot C \cdot A$, where $A$, $B$ and $C$ are matrices expressed as arrays?

# Function

```
float square(float x) {
        float powered = x * x;
        return powered;
}
//...
        float distance = square(x - a) + square(y - b);
```

- There is no keyword that defines a function
- Function declaration starts with the type it returns, then there's its name followed by types of arguments it takes and it ends by a block of code
- Arguments have to be named in order to be used in the function
- The value behind the `return` keyword is the value returned by the function

# Function #2

```
std::string even(int x) {
        x = x % 2;
        if (x == 1)
                return "no";
        else
                return "yes";
}
```

- There can be any number of `return` statements, the first one the execution reaches exits the function
- Variable `x` is a copy of the argument the function received, changing it has no outside effect
- Two functions can have the same name as long as they have different argument types

# Function #3

```
void swapVars(float& x, float& y) {
        float orig = x;
        x = y;
        y = orig;
}
```

- The ampersand after the variable type (&) makes it a reference
- Editing a variable through a reference changes the variable used to call the function
- The `void` keyword means that no variable is returned and `return` is unnecessary

# Function #4

```
float sum(const std::vector<float>& vec) {
        float total = 0;
        for (float val : vec)
                total += val;
        return total;
}
```

- References are mainly used to prevent copying large objects that would take a lot of time (such as containers, strings, ...)
- `const` is a modifier that prevents a variable from being edited; it's useful to mark you don't want to edit it and you will not be able to edit it accidentally

# Exercise

1. Write a function that checks if $x$ is divisible by $y$ and use it in an interactible program
2. Write a function that returns the average of numbers in a vector
3. Write a function that appends one vector at the end of another
4. Write a function that computes matrix multiplication
5. Calculate $A \cdot (B \cdot A) \cdot C \cdot A$, where $A$, $B$ and $C$ are matrices expressed as `easy::vector<easy::vector<float»` (or `std::vector`)

# Operators

```
float operator^(float num, const std::string& num2) {
        return pow(num, std::stof(num2));
}
```

- Operator is a function called as an operation
- It's a normal function, it just may be called more conveniently

```
float x = y ^ std::string("3");
```

- Operators can be defined only if one of the arguments isn't a primitive type (string, array, vector, ...)

# Generic functions

```
template<typename T>
T sum(const std::vector<T>& vec) {
        T total = 0;
        for (T current : vec)
                total += current;
        return total;
}
```

- Can be applied on any vector, but will not compile if the type in vector can't be set to 0 or summed
- The type `T` is determined based on arguments when compiling
- If necessary, the type of `T` can be specified

```
float sum = sum<float>(theVector);
```

# Lambdas

```
float x = 2;
auto func = [&] (int y) {
        x = (x + y) * 2;
};
func(y);
```

- Lambda functions are local functions that are stored in variables
- A lambda can have access to variables in its context
- Use [=] instead of [&] to use copies of the variables (it's necessary if the lambda outlives these variables)

# Exercise

1. Write a function that sums numbers in two vectors of the same length, returning the vector of results (vector sum from algebra)
2. Change the function so that it could be called using the + operator

Advanced exercices:

1. Create a `cube` function that calculates third power of any type of numeric variable (without using `pow`)
2. Write a vector-summing function that can be used on vectors of any numeric type (but both have the same type)
3. Change the function so that it could be called using the + operator

# Homework

- Write a vector subtracting function that removes all occurrences of elements in the vector in second argument (in sequence) from the vector in the first argument
- You have two weeks to do it

# Advanced Homework

- Create an operator * that can be used to multiply matrices of type std::array<std::array<T,S1>,S2>
- You have two weeks to do it

```
template<typename T, int S>
T sum(const std::array<T, S> array) {
        T total = 0;
        for (int i = 0; i < S; i++)
                total += array[i];
        return total;
}
```