HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Introduction to Programming

Procedural Decomposition

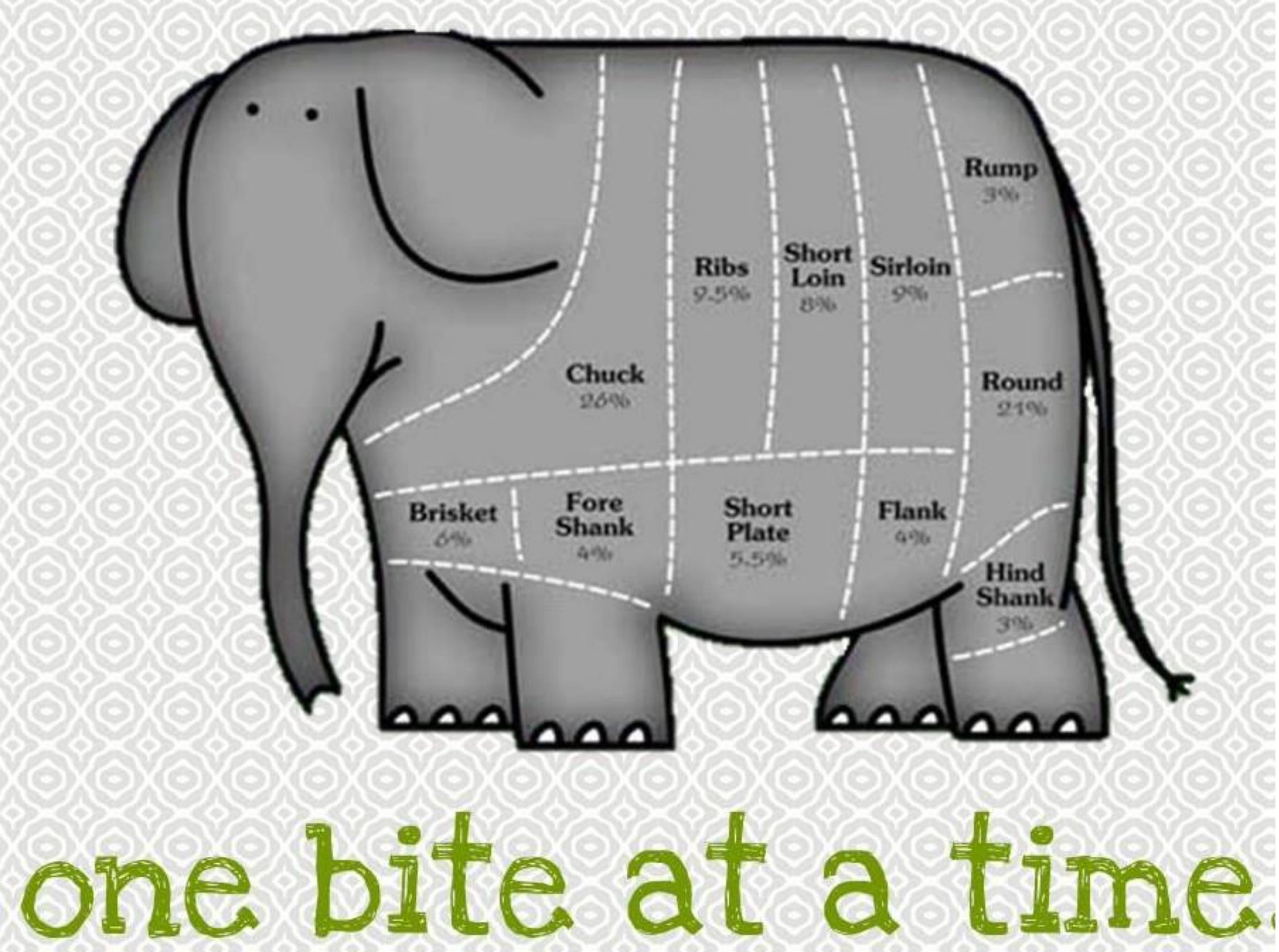
Sergey Shershakov

#5/24 Jan 2019

Test 3 (5 pts)

https://goo.gl/forms/YkmXxRvIsbkxSOoC2

What is Decomposition?



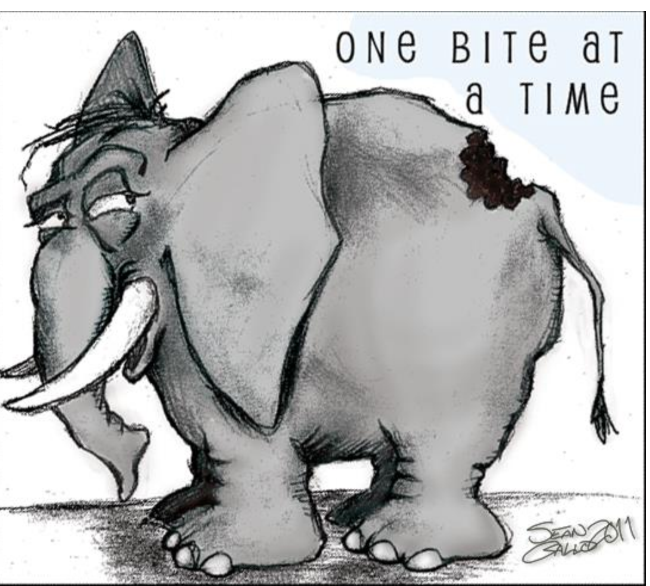
What is Decomposition?

- Contest (homework) 2 Problem 3: **Transpose rectangular matrix**

https://blog.jetbrains.com/idea/2017/09/code-smells-too-many-problems/

What is Decomposition?

- Contest (homework) 2 Problem 3:
  Transpose rectangular matrix
- Break down the problem into smaller subparts
  - Break down each subpart into smaller subsubpart
    - Break down each subpart into smaller subsubsubpart
      - Try to repeat splitting until an individual piece of the problem becomes a piece of cake



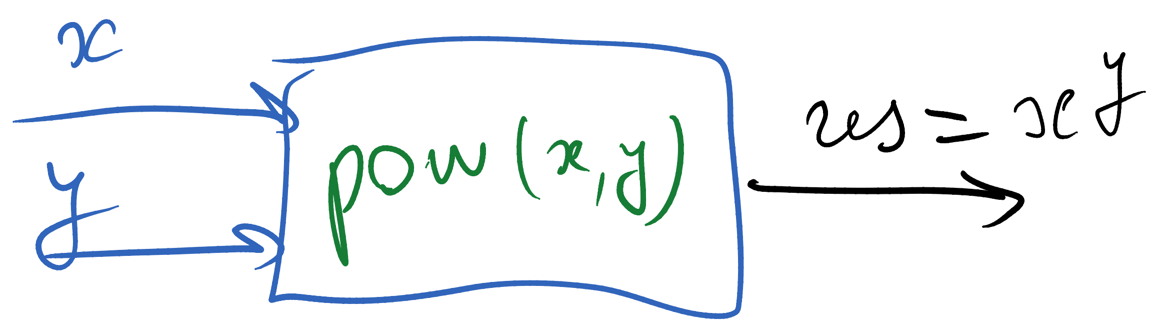
What is Decomposition?

- How to transpose a matrix from Problem 3?
  1. create an object representing the matrix;
  2. read the elements of the matrix, row by row, from `std::cin`;
  3. print the elements of the initial matrix to `std::cout`;
  4. make a transposition:
     a) create another matrix object for the transposed matrix;
     b) read the initial matrix, row by row, and copy elements to the transposed matrix, column by column
  5. print the elements of the transposed matrix to `std::cout`.

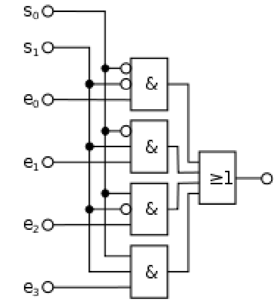
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}^{\mathrm{T}}$$

Functional Decomposition in C++

- *Functions* are modules from which C++ programs are built.
- A *function*: ← method
  - has input in the form of parameters
  - performs some operations taking into account input parameters
  - returns a result
- A *procedure*, *subroutine*, *subprograms* are functions with no return value (indicated by the `void` keyword)



Free and Member Functions



- *Free Function* (non-member function) is a function defined in the global scope or in a narrowed scope of a namespace
  - acts like a combinational scheme:
  - the output of a scheme is determined only by its input (if no global objects are involved)

```
int mult(int x, int y)
{
    return x * y;
}
//…
int res = mult(2, 3);
```

- *Member* (of a class/structure) is a function defined in the scope of a class/structure and using the state of an individual instance of the class/structure

```
class string {
  // ...
    // member function
    size_t size() const { ... }
  // ...
};

string s1 = "Abc";
s1.size() == 3;

string s2 = "Hello, world!";
s2.size() == 13;
```

(Free) Function Definition

return value

function name

list of formal parameters

void

```
int mult(int x, int y)
{
    int res = x * y;

    return res;
}
```

function header

function definition (body)

terminates function and returns specific value

return;

A Caller, a Callee and a Prototype

```
int mult(int x, int y)
{
    return x * y;
}
```

callee

```
int main()
{
    int x, y;
    cin >> x >> y;

    int res = mult(x, y);
    cout << res;
}
```

caller

```
int main()
{
    int x, y;
    cin >> x >> y;

    int res = mult(x, y);
    cout << res;
}


int mult(int x, int y)
{
    return x * y;
}
```

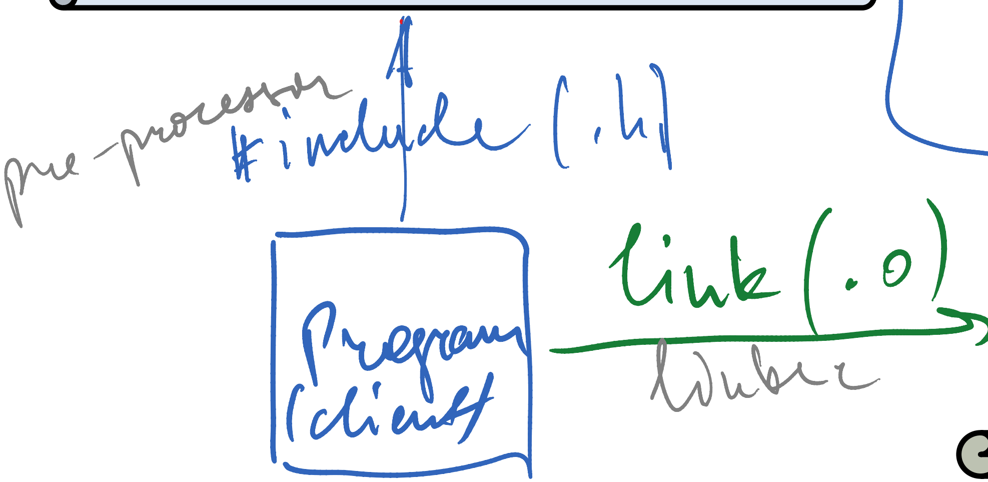
Function: Interface and Implementation

- **Interface**: `myfunc.h`

```
// myfunc.h

int mult(int x, int y);
double div(double x, double y);
double pow(double x, int y);

//...
```

- **Implementation**: `myfunc.cpp`

```
// myfunc.cpp

int mult(int x, int y)
{
    return x * y;
}

double div(double x, double y)
{
    return x / y;
}

double pow(double x, int y)
{
    double res = 1;
    for(int i = 0; i < y; ++i)
        res *= x;
    return res;
}
```



Formal Parameters

- *Formal Parameters* are a list of parameters defined in a function's definition

```
double pow(double x, int y)

pow(double , int )
```

Return Value

```
double div(double x, double y)
{
    if(y == 0)
        return std::numeric_limits<double>::infinity();

    return x / y;
}
```

```
void print(string& s)
{
    for(char ch : s)
        cout << '\'' << ch
             << "\', ";
}
```

```
void print(string& s)
{
    if(s.size() == 0)
        return;

    for(char ch : s)
        cout << '\''
             << ch << "\', ";
}
```

Formal and Actual Parameters

```
double pow(double x, int y) {…}

//...

double r1 = pow(1.23, 5);

double r2 = pow(2. ,  mult(2, 3));

double r3 = pow(3  ,  1 + 1);

double r4 = pow(pow(2, 3), 2);
```

Function Overloading

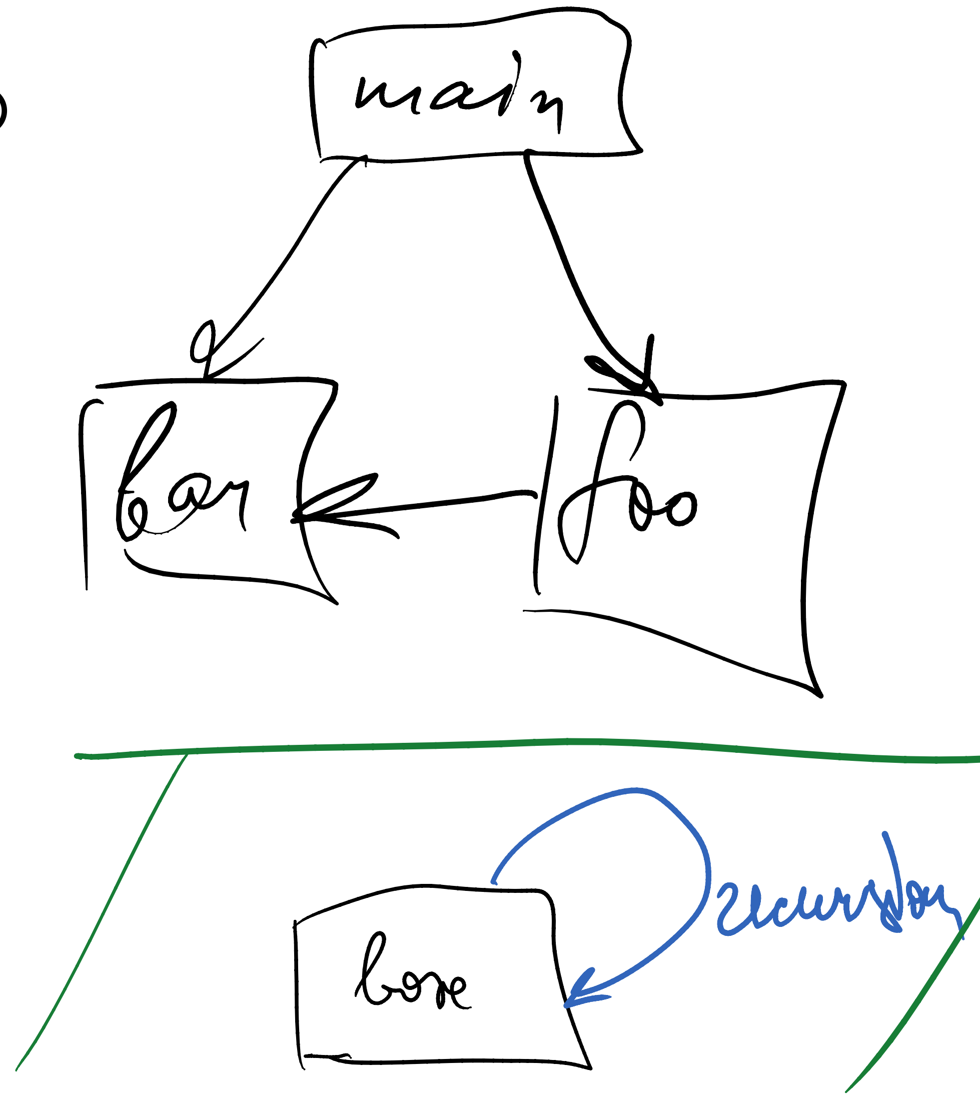
- *Function Overloading* is defining a set of functions with the same name:
  - signatures must be different! →
    - types of parameters must be different

```
int          pow(int x, int y);
double       pow(double x, int y);
long         pow(long x, int y);
long long    pow(long x, int y);
```

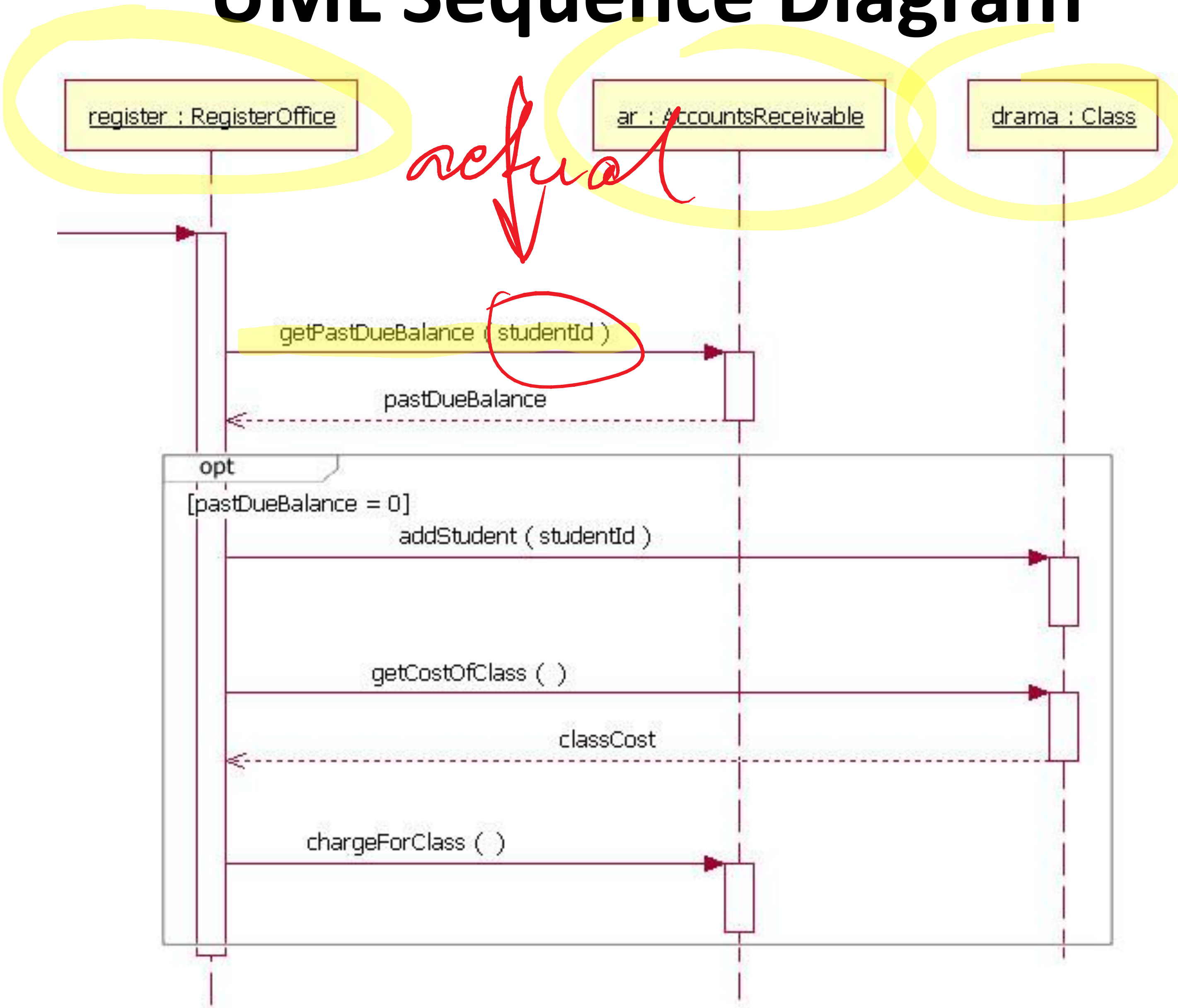
- What about return value?

Function Call Diagram

```
double bar(double p)
{
    ...
}


int foo(int x, int y)
{
    double r = (PI * x) / y
    return (int)bar(r);
}


int main()
{
    //...
    int r = foo(x, 17);
    r += bar(42.);
}
```



UML Sequence Diagram



[https://www.ibm.com/developerworks/rational/library/3101.html]

REFERENCE TYPES

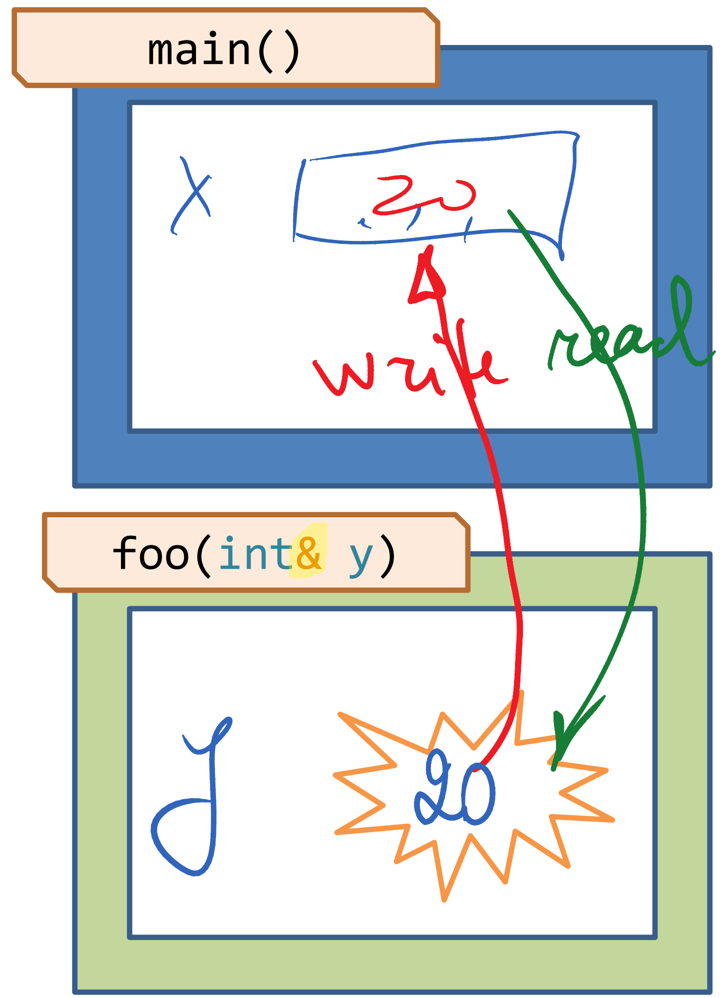
Reference

Mole hole, wormhole

- Reference is another name for an object.
- Reference type:

  `typename&`

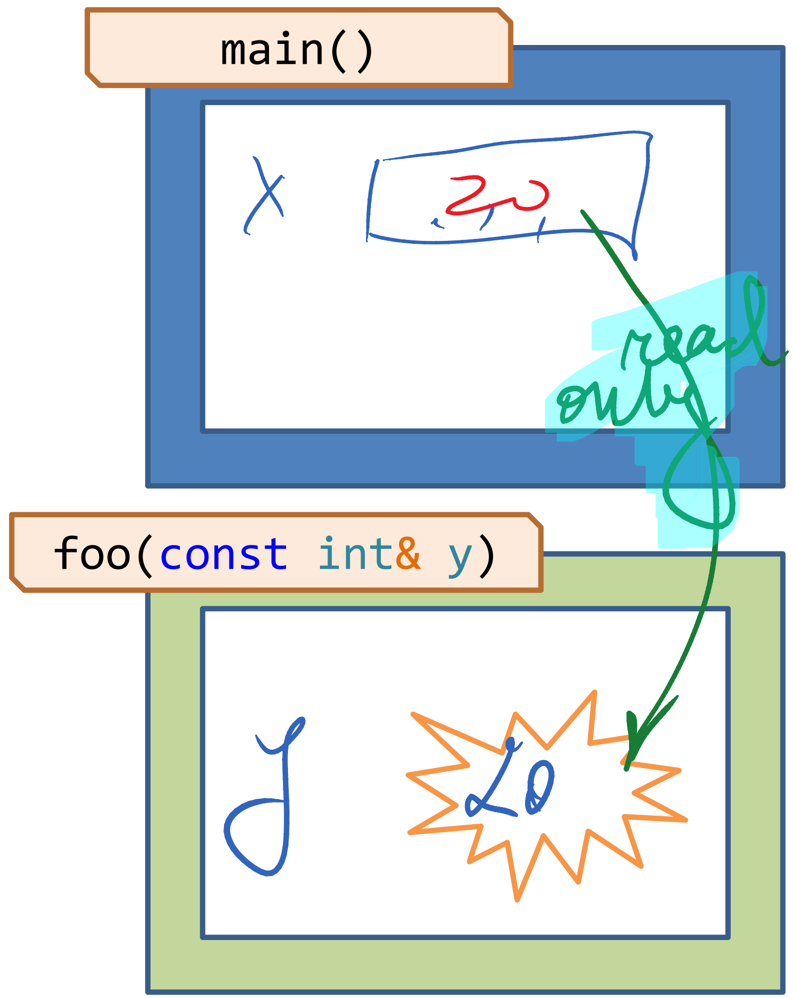
```
int x = 10;
cout << x;     // 10

int& x1 = x;   // binding
cout << x1;    // 10


x1 = 15;
cout << x1;    // 15
cout << x;     // 15


x = 42;
cout << x;     // 42
cout << x1;    // 42
```



Const Reference

- Reference is another name for an object.
- Reference type:

  `const typename&`



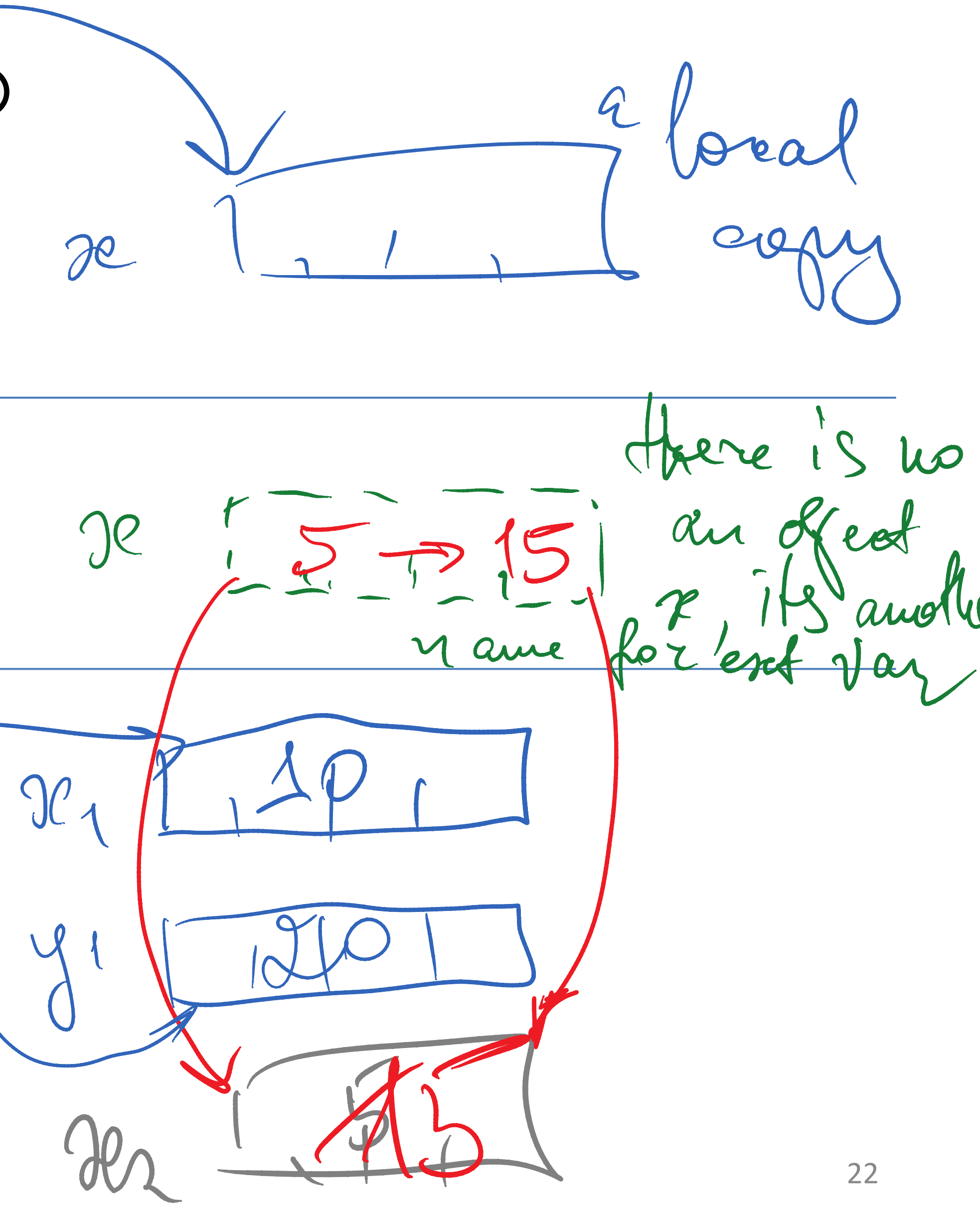
The Range-Based for Loop (Ref. Version)

- Iterates over a collection of elements from the first to the last.
- Can modify a collection by using **reference type** (will get back to this feature later)

```
double koefs[] = {1.12, 2.13, 3.14, 4.15, 5.16};

for (double x : koefs)
    cout << x << std::endl;

for (int x : {1, 1, 2, 3, 5})
    cout << x << " ";

for (double& y : koefs)
    y = y * 2;     // doubles the values
```

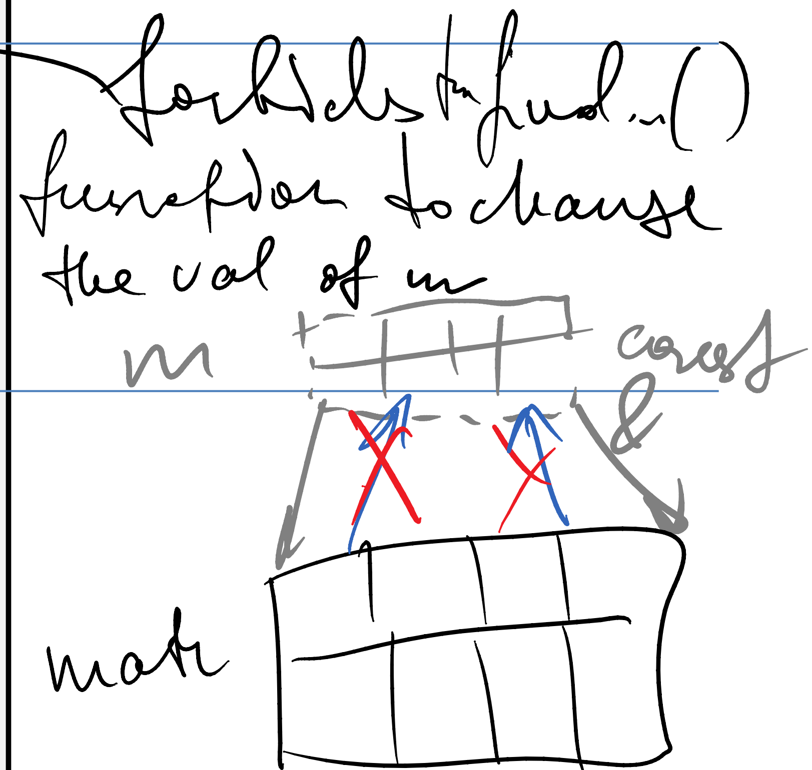
Value and Reference Parameters of a Function

```
int dblIt(int x)
{
    x = x * 2;
    return x;
}

void tripleIt(int& x)
{
    x = x * 3;
}

int main()
{
    int x1 = 10;
    int y1 = dblIt(x1);   // 20
    x1;                   // 10


    int x2 = 5;
    tripleIt(x2);
    x2;                   // 15
}
```



Returning Multiple Values from a Function

```
typedef .....   Matrix;

Matrix createMatrix(int m, int n) { /*...*/ }

void findMinEl(const Matrix& m, int& i, int& j)
{   //...
    for(i = 0...
        for(j = 0...
            if(...)  return;
}

int main()
{
    //...
    Matrix matr = createMatrix(5, 8);

    // indices of the first min element
    int i, j;
    findMinEl(matr, i, j);

    return 0;
}
```



STD::VECTOR

std::vector as a Dynamic 1-d Array

```
#include <vector>

std::vector<int> v1;
v1.size();

v1.push_back(10);
v1.push_back(20);
v1.push_back(30);
v1.size();


int a = v1[0];
a = v1[1];
a = v1[2];
a = v1[3];

int b = v1.at(0);
b = v1.at(1);
b = v1.at(2);
b = v1.at(3);

v1.resize(5);

v1.resize(2);
```

The typedef Keyword for Creating Aliases for Complex Types

```cpp
#include <vector>

std::vector<std::string> lines;

void printStrVector(const std::vector<std::string>& lines)
{
    //...
}

std::vector<std::string> readStrVector(size_t n)
{
    //..
}
```

```cpp
typedef   std::vector<std::string>  IntVector;
```

```cpp
typedef std::vector<std::string> IntVector;

IntVector lines;

void printStrVector(const IntVector& lines) { /* ... */ }

IntVector readStrVector(size_t n) { /* ... */ }
```

Defining a Matrix as a Vector of Vectors

```cpp
#include <vector>

typedef std::vector< std::vector<int> > IntMatrix;

IntMatrix m1;


// represents individual rows
typedef std::vector<std::string> IntVector;
```