



HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY



Introduction to Programming

Procedural Decomposition

Sergey Shershakov

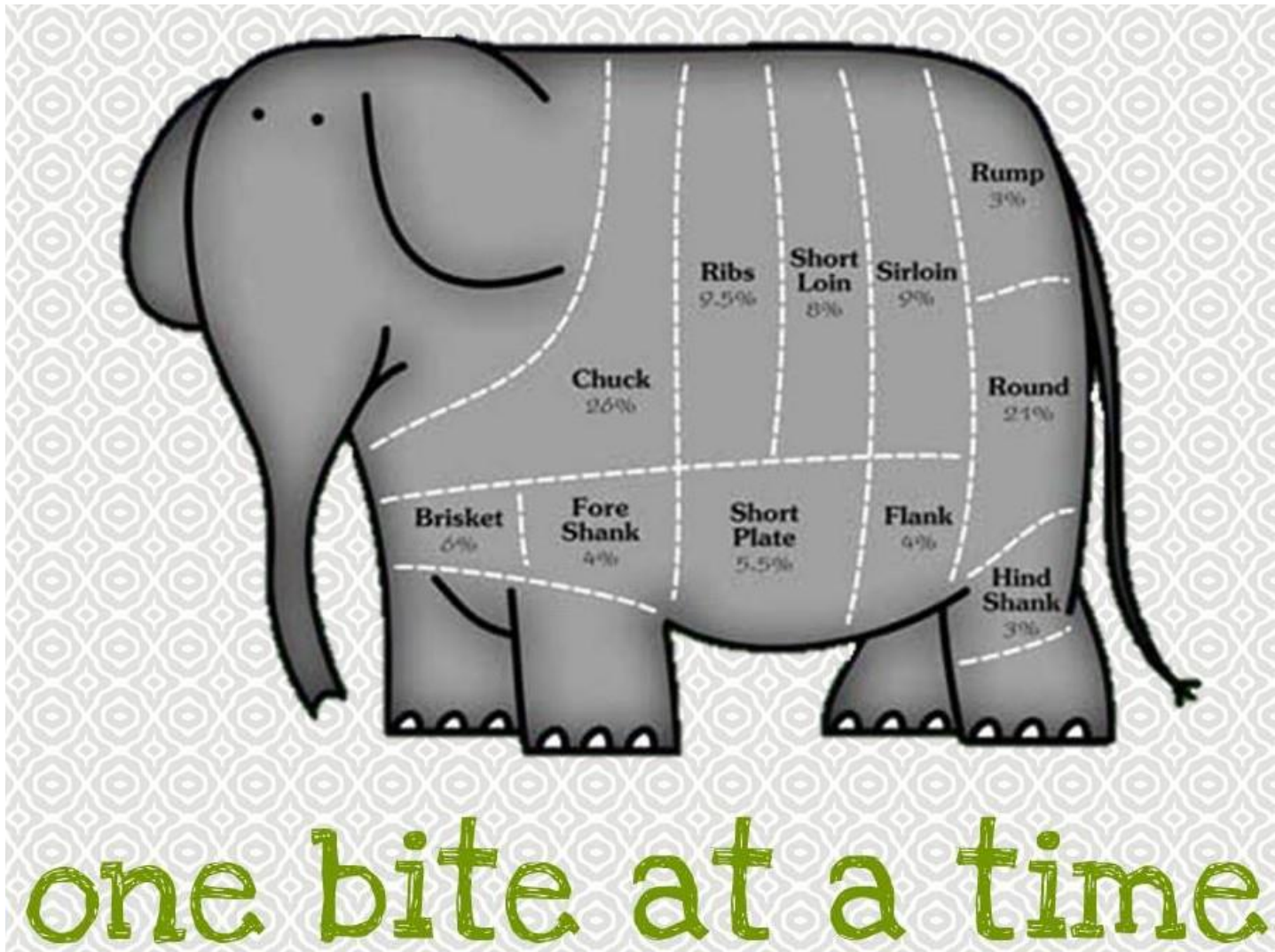
#5/24 Jan 2019

Test 3 (5 pts)



<https://goo.gl/forms/YkmXxRvIlsbkxSOoC2>

What is Decomposition?



What is Decomposition?

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

```
41 @ static MappedField validateQuery(final Class clazz, final Mapper mapper, final StringBuilder origProp, final FilterOperator op, final
42 MappedField mf = null;
43 final String prop = origProp.toString();
44 boolean hasTranslations = false;
45 if (!origProp.substring(0, 1).equals("$")) {
46     final String[] parts = prop.split(regex: "\\.");
47     if (clazz == null) { return null; }
48     MappedClass mc = mapper.getMappedClass(clazz);
49     //CHECKSTYLE:OFF
50     for (int i = 0; ; ) {
51         //CHECKSTYLE:ON
52         final String part = parts[i];
53         boolean fieldIsArrayOperator = part.equals("$");
54         mf = mc.getMappedField(part);
55         //translate from java field name to stored field name
56         if (mf == null && !fieldIsArrayOperator) {
57             mf = mc.getMappedFieldByJavaField(part);
58             if (validateNames && mf == null) {
59                 throw new ValidationException("The field '%s' could not be found in '%s' while validating - %s; if you wish to search/update into @Reference/@Serialized fields");
60             }
61             mf = mc.getMappedField(part);
62             if (mf != null) {
63                 parts[i] = mf.getNameToStore();
64             }
65             i++;
66             if (mf != null && mf.isMap()) {
67                 //skip the map key validation, and move to the next
68                 i++;
69             }
70             if (i >= parts.length) {
71                 break;
72             }
73             //get the next MappedClass for the next field validation
74             mc = mapper.getMappedClass((mf.isSingleValue()) ? mf.getType() : mf.getSubClass());
75         }
76         //validateNames && !canQueryPast(mf) {
77         throw new ValidationException(format("Cannot use dot-notation past '%s' in '%s'; found while validating - %s", part, prop, origProp));
78     }
79     if (mf == null && mc.isInterface()) {
80         break;
81     } else if (mf == null) {
82         throw new ValidationException(format("The field '%s' could not be found in '%s'", prop, mc.getClazz().getName()));
83     }
84     //get the next MappedClass for the next field validation
85     mc = mapper.getMappedClass((mf.isSingleValue()) ? mf.getType() : mf.getSubClass());
86 }
87 }
88
89 //record new property string if there has been a translation to any part
90 if (hasTranslations) {
91     origProp.setLength(0); // clear existing content
92     origProp.append(parts[0]);
93     for (int i = 1; i < parts.length; i++) {
```

What's a prop?

What's a part?

Eek!

No!
Why all the null checks?
Control the loop

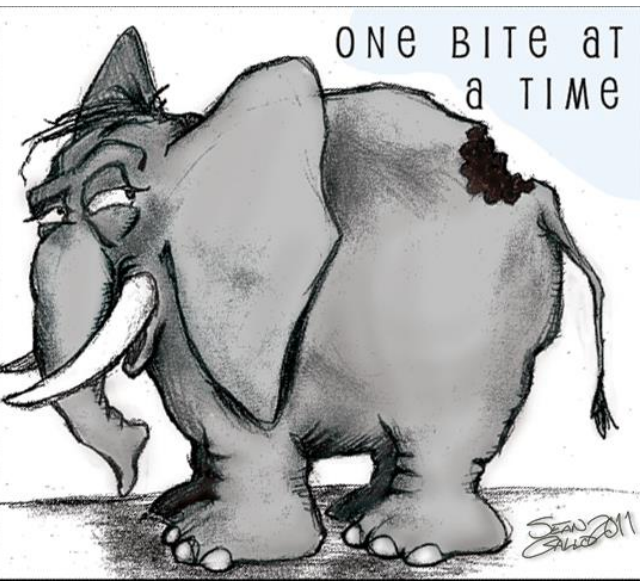
Comments, because code is unclear

Parameter mutation!

<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



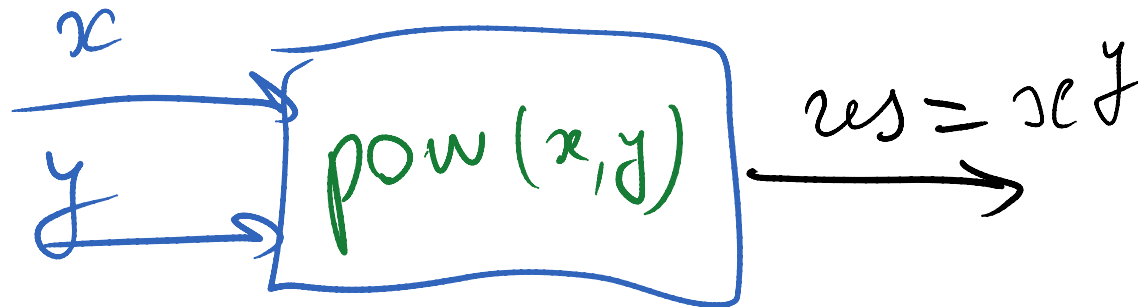
What is Decomposition?

- 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`.

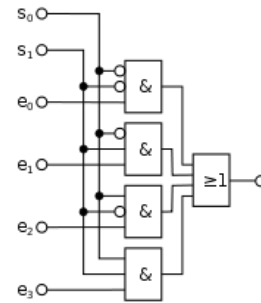
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}^T$$

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);
```

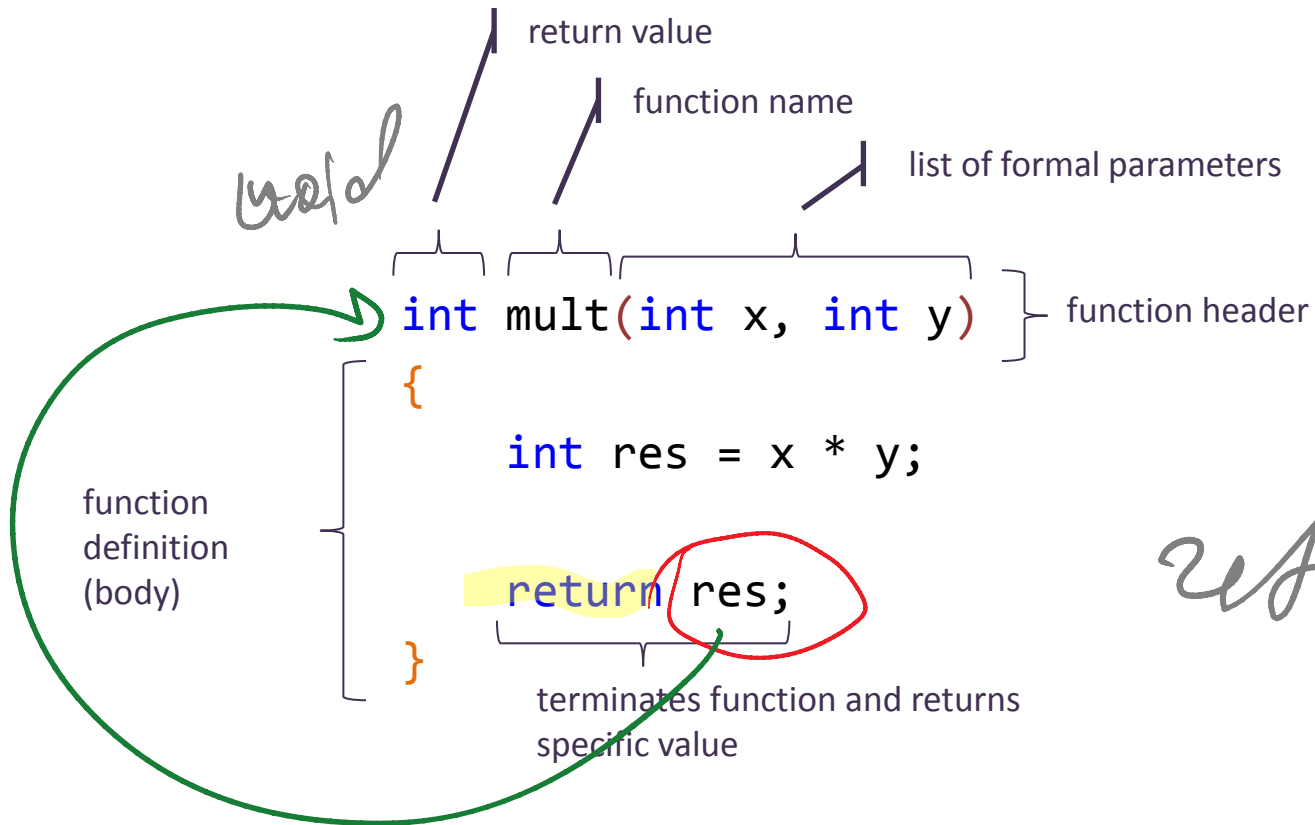
- *Member* (of a class/structure) is a function defined in the scope of a class/structure and using the state of an individual instance of the class/structure

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

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

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


(Free) Function Definition



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`

- Implementation: `myfunc.cpp`

```
// myfunc.h

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

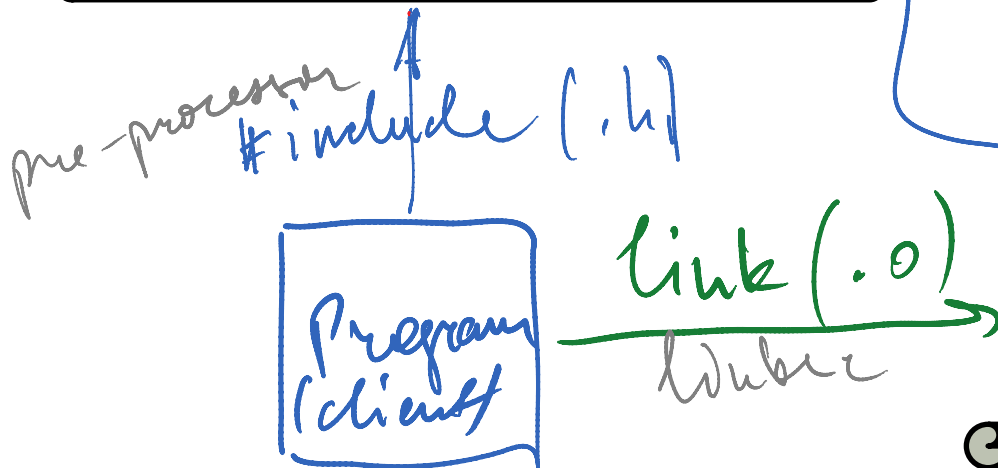
//...
```

```
// myfunc.cpp

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

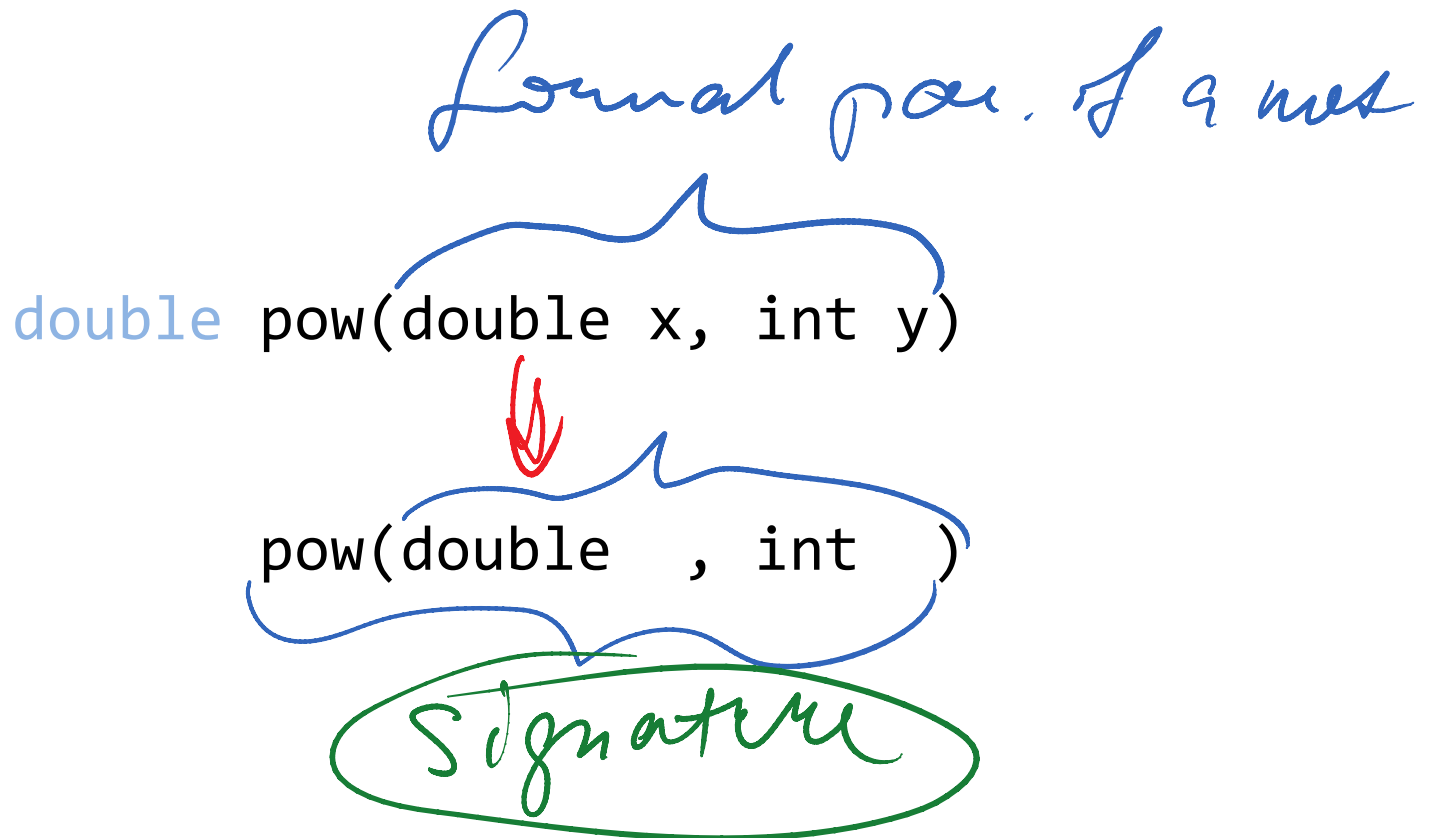
double div(double x, double y)
{
    return x / y;
}

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



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 *formal* 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);
```

*only have
return type
different*

*can't
distinguish*

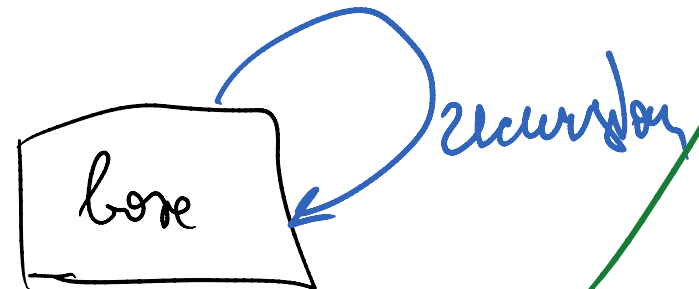
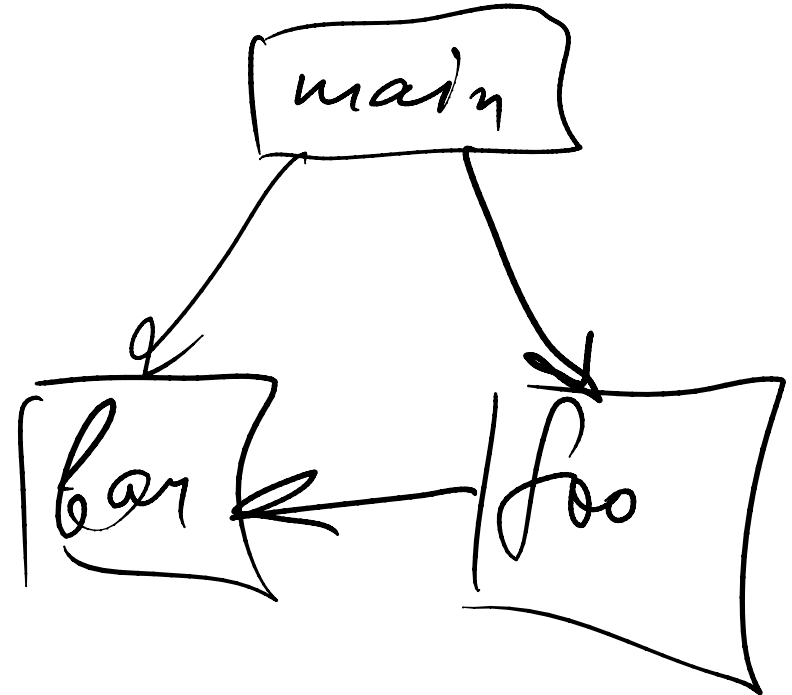
- What about return value?

Function Call Diagram

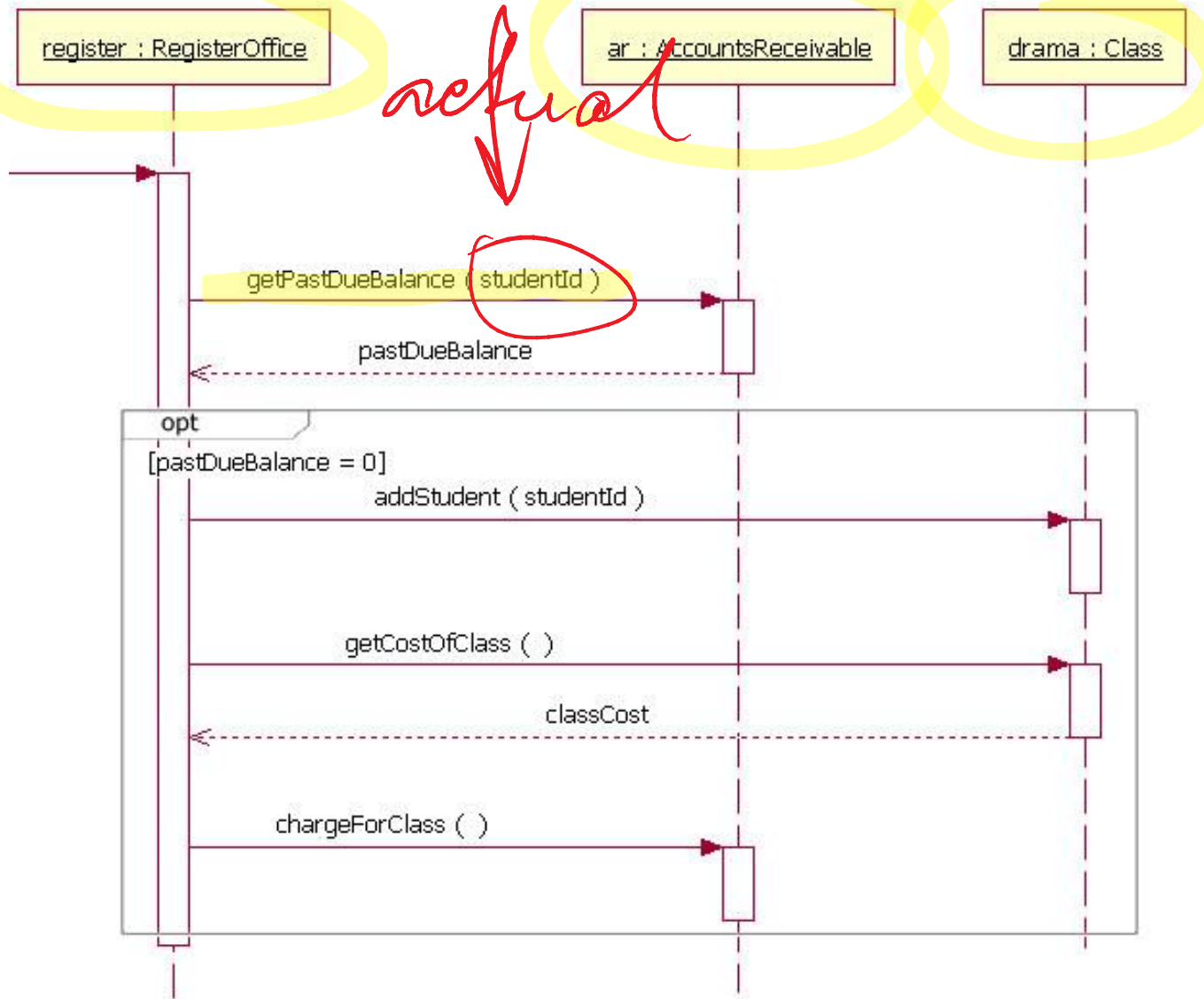
```
double bar(double p)
{
    ...
}

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



REFERENCE TYPES

Reference



Mole hole,
wormhole

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

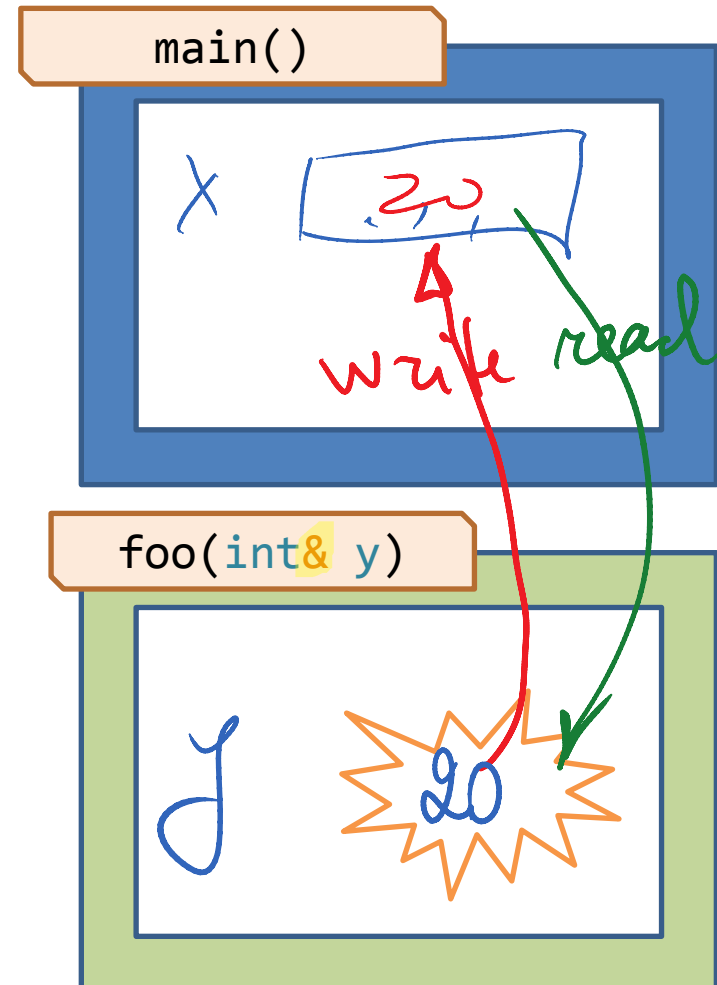
`typename&`

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

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

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

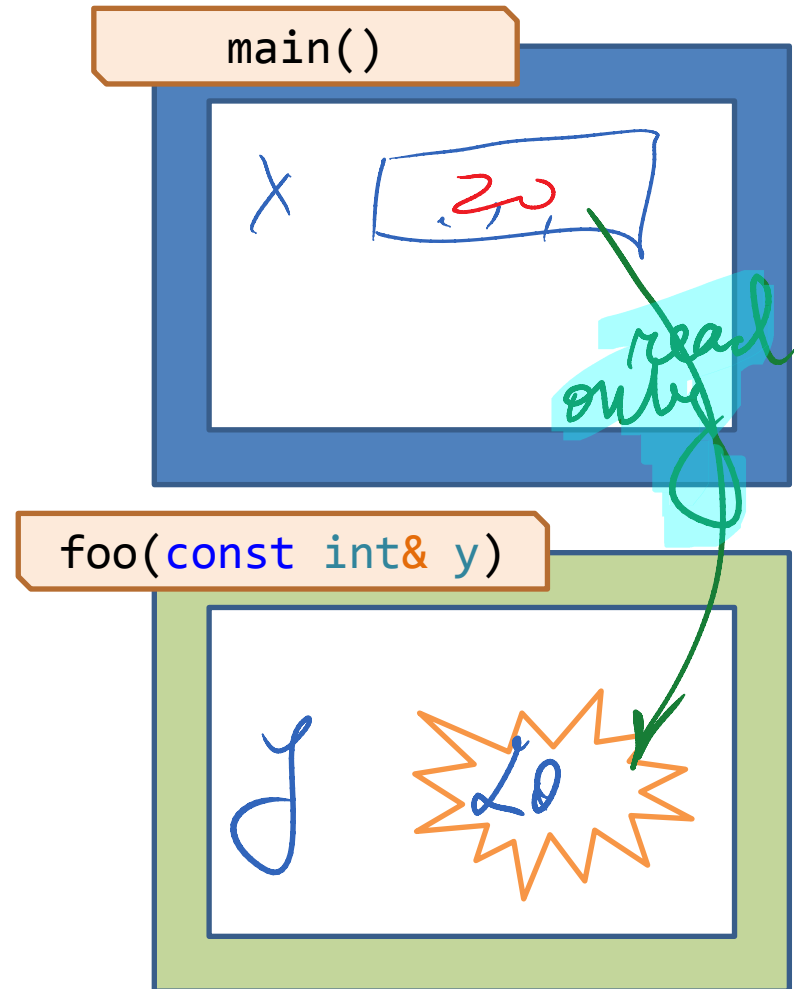
x = 42;
cout << x;    // 42
cout << x1;   // 42
```



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

```
for (double x : koeffs)  
    cout << x << std::endl;
```

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

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

Handwritten annotations in red:

- 2.24
- 4.26
- 6.28
- An upward-pointing arrow above 6.28
- Red lines connecting the annotations to the corresponding elements in the array {1.12, 2.13, 3.14, 4.15, 5.16}.

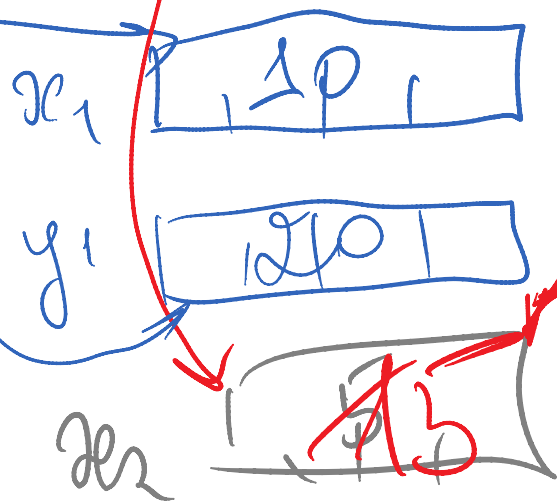
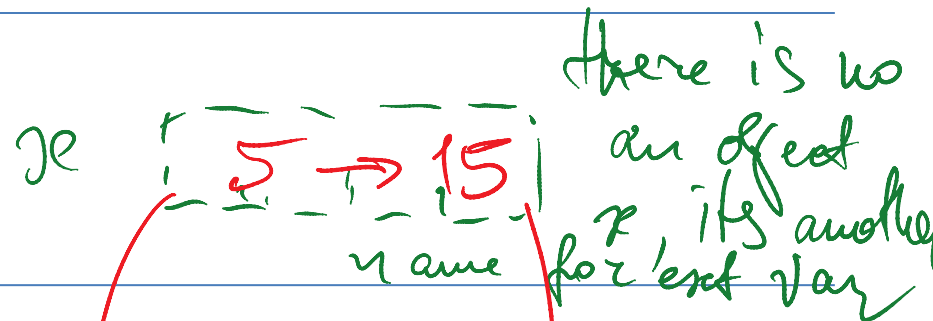
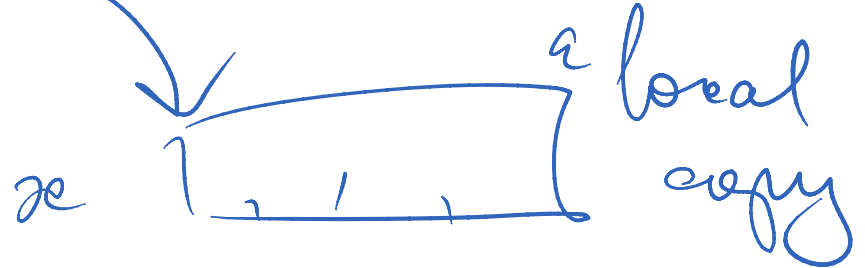
Value and Reference Parameters of a Function

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

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

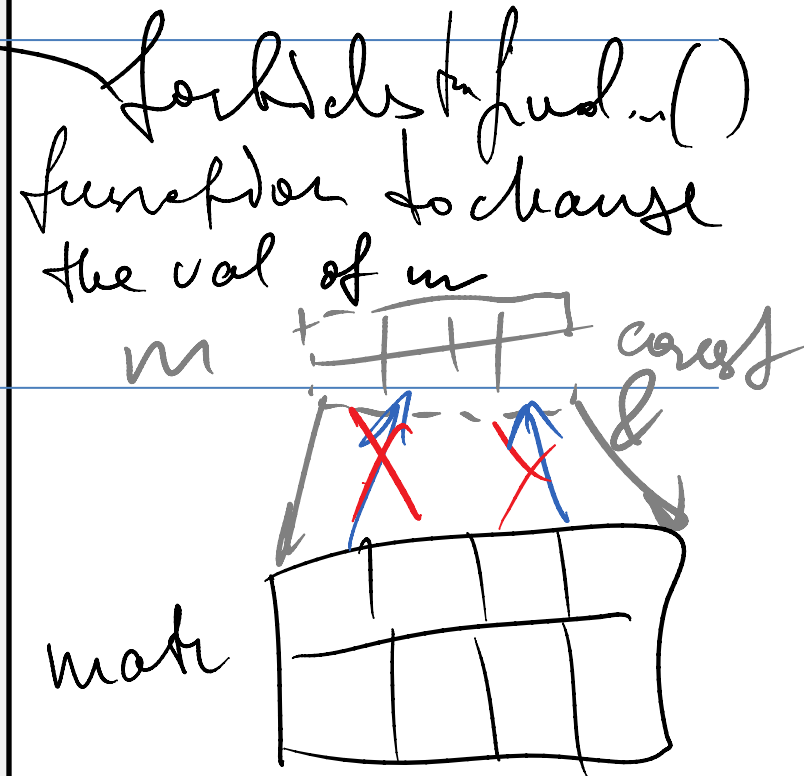
```
int main()
{
    int x1 = 10;
    int y1 = dblIt(x1); // 20
    x1; // 10

    int x2 = 5;
    tripleIt(x2); // 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

```
#include <vector>

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

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

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

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

```
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

```
#include <vector>

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

IntMatrix m1;

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

