HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Introduction to Programming

From Structures to Classes

Sergey Shershakov

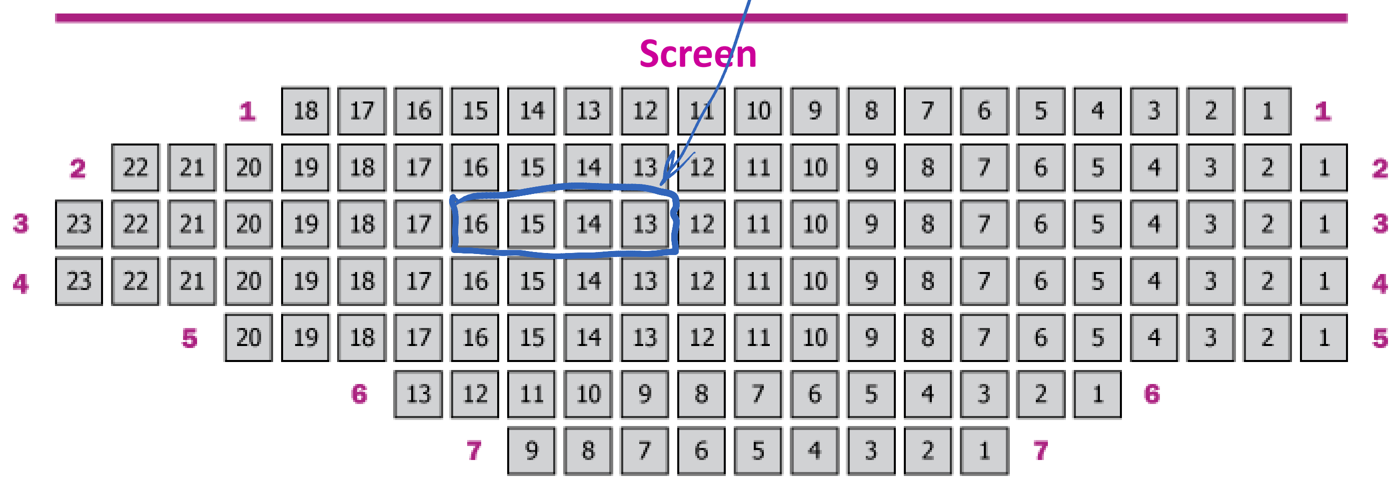
#8, #9/5, 7 Feb 2019

On the Intermediate Test

- A big “Kontrolyanya Rabota” is planned during the week beginning on Feb 18 (Feb 19 or Feb 21)
- Duration is 1 class (2 ac. units)
- A personal laptop is needed:
  - for those who are not able to bring their own, a computer class will be booked;
  - we need to count heads (there will be a poll).

Let's Go to the Cinema!

13(12)



2

(2;12)

Jagged Array

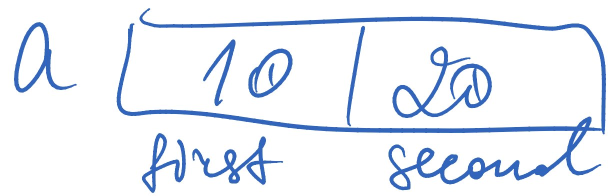
1) input data: m rows, n_i seats for each i-th row; 1 — the seat is sold, 0 — the seat is free;
2) print data in a different format: a row per line, * is for sold seats, . is for free; sold/total ratio in the end of each row/line;
3) someone would like to buy k adjacent seats in the same row; one needs to determine whether it is possible or not;
4) how to modify the printing method for highlighting the free k seats by using "XXXX" notation?

The `std::pair` Utility Class

- Simple structure representing a pair of objects that can have a different type

```
std::pair<Type1, Type2>
```

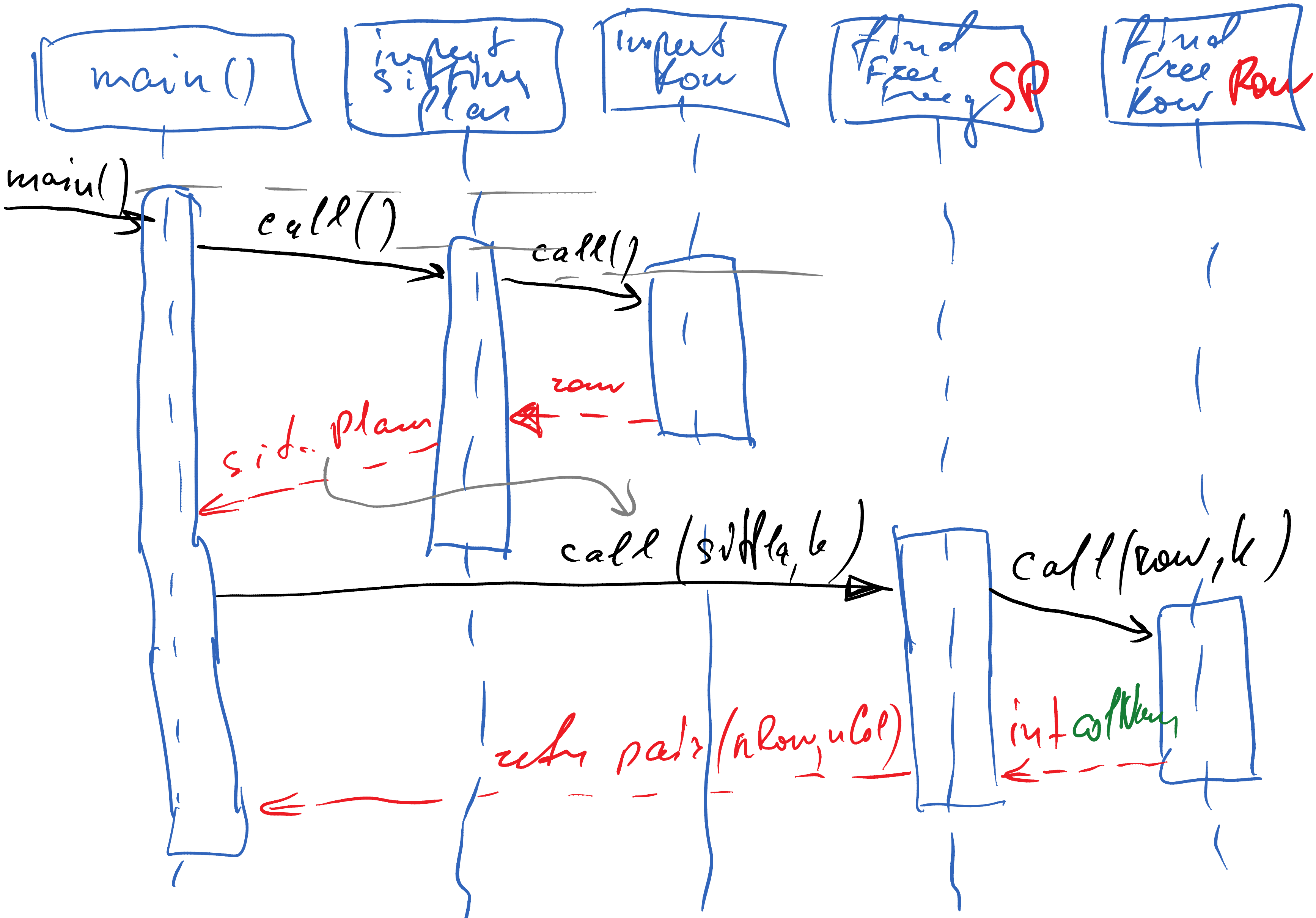
```
pair<int, int> a(10, 20);
a.first == 10;
a.second == 20;
```




```
return {i, freeCol};
return std::make_pair(i, freeCol);
return std::pair<int, int>(i, freeCol);
```

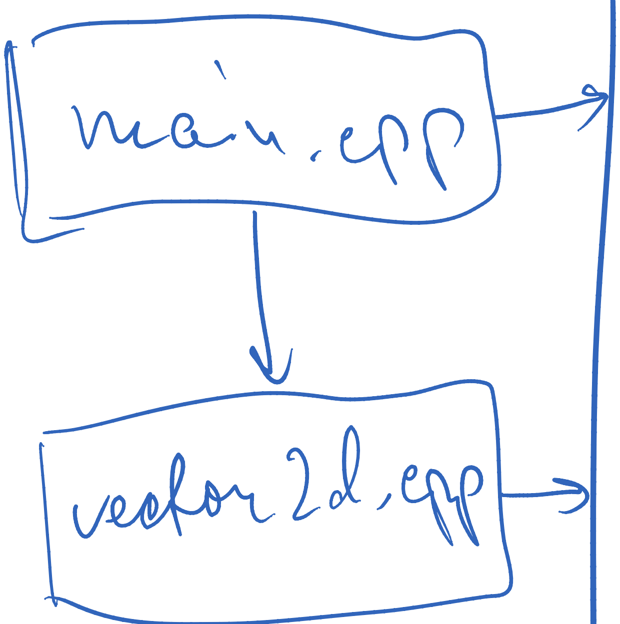
http://www.cplusplus.com/reference/utility/pair/pair/
https://en.cppreference.com/w/cpp/utility/pair

UML Sequence Diagram of Calling Functions



INTRODUCTION TO OOP

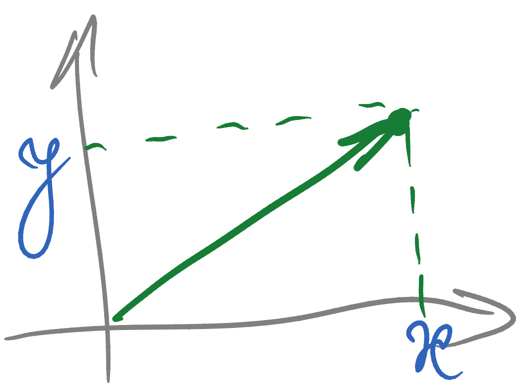
Vector2d Structure



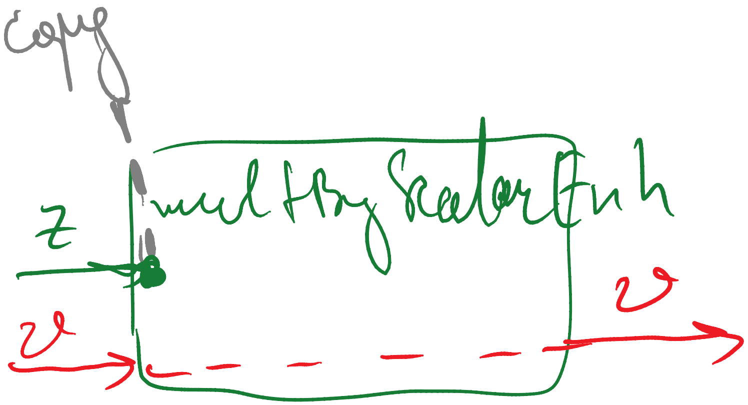
vector2d.h

vector2d.h*   <Select Symbol>

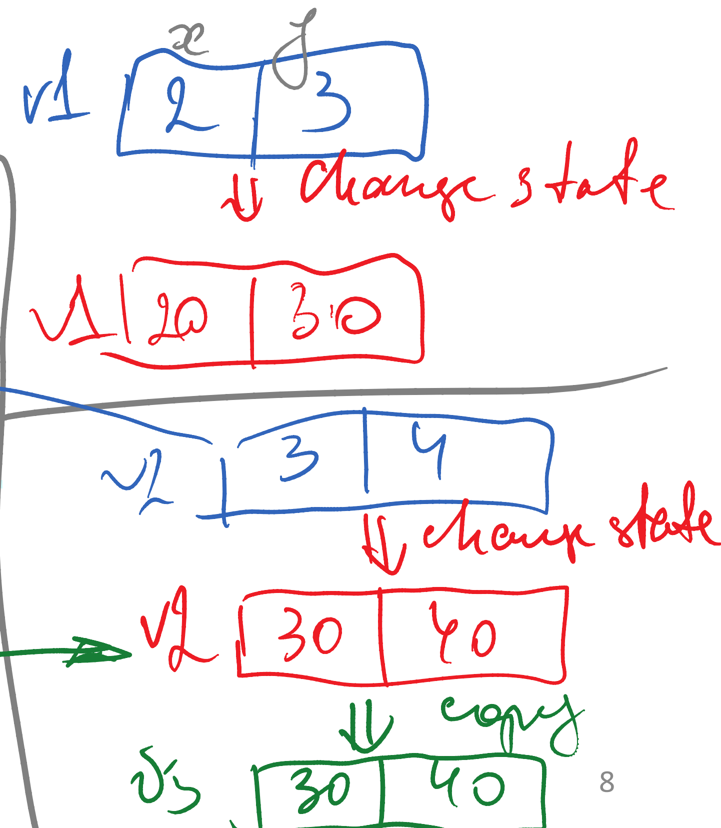
```cpp
/*! \file vector2d.h
 *  Definition of the structure Vector2d.
 */

#ifndef VECTOR2D_H
#define VECTOR2D_H


struct Vector2d
{
    double x;
    double y;
};


#endif // VECTOR2D_H

```



Passing-through of an Object by Reference




```
int main()
{
    Vector2d v1 = {2, 3};
    Vector2d v2 = {3, 4};

    multByScalar(v1, 10);
    Vector2d v3 = multByScalarEnh(v2, 10);
```




```
void multByScalar(Vector2d& v, double z)
{
    v.x *= z;
    v.y *= z;
}

Vector2d& multByScalarEnh(Vector2d& v, double z)
{
    v.x *= z;
    v.y *= z;

    return v;
}
```

Output Vector2d to a Stream

```
28        std::cout << "v1: " << v1 << '\n';
```

```
F:\HSE\training\DSBA\programming\programs\lecture08\src\ex_2\ex_2.cpp:28:28: note:   cannot convert
'v1' (type 'Vector2d') to type 'const std::error_code&'
     std::cout << "v1: " << v1 << '\n';
                            ^
```

- One needs to “teach” the compiler how to output objects of a custom type:
  - overload `operator<<` for the `std::ostream` type:

```
std::ostream& operator<<(std::ostream& s, const Vector2d& v)
{
    s << '(' << v.x << ", " << v.y << ')';

    return s;
}
```

- Why do we need to return the stream? why it is by reference?

Problem: Calculations of a Vector's Length

- We don't want to recalculate a vector's length until its coordinates, x and y, are not changed
  - cache the length value as a separate field;
  - treat a negative value as a sign that no length has been calculated previously;

```
struct Vector2d
{
    double x;
    double y;

    double length;
};
```

```
double calcLength(/*const */Vector2d& v)
{
    // if the value has not been calculated previousl
    if(v.length < 0)
        v.length = sqrt(v.x * v.x + v.y * v.y);

    return v.length;
}
```

- Possible problems:
  - how to initialize the `length` field before the very first use?
  - how to guarantee that `length` value will be invalidated when either x or y is changed?

Putting Data and Behavior Together

```
struct Vector2d
{
    double x;
    double y;

    double length;
};
```

determines the state of an object

determines the behavior of an object

```
void multByScalar(Vector2d& v, double z);

Vector2d& multByScalarEnh(Vector2d& v, double z);

double calcLength(/*const */Vector2d& v);
```

- Vector2d is passed as a parameter, v, to all of these methods;
  - combine them together in a more natural way!

Putting Data and Behavior Together

```
struct Vector2d
{
    //----< Fields >----

    double x;
    double y;
    double length;                                          ///< Stores the 

    //----< Methods >----

    void multByScalar(/*Vector2d& v, */ double z);          ///< Multiplicat
    Vector2d& multByScalarEnh(/*Vector2d& v, */ double z);  ///< Enhanced mu
    double calcLength(/*Vector2d& v*/);                     ///< Length calc

}; // struct Vector2d
```

Putting Data and Behavior Together

```
struct Vector2d
{
    //----< Fields >----

    double x;
    double y;
    double length;                                   ///< Stores the cached va


    //----< Methods >----

    void multByScalar(double z);                     ///< Multiplication.
    Vector2d& multByScalarEnh(double z);             ///< Enhanced multiplicat
    double calcLength();                             ///< Length calculation.

}; // struct Vector2d
```

How to Implement Methods of a Structure?

- Where to put? — ….. Vector2d.cpp

```
void Vector2d::multByScalar(/* Vector2d& v, */ double z)
{
    /* v.*/ x *= z;
    /* v.*/ y *= z;
}

double Vector2d::calcLength(/* Vector2d& v */)
{
    // if the value has not been calculated previously...
    if(/*v.*/length < 0)
        /*v.*/length = sqrt(/*v.*/x * /*v.*/x + /*v.*/y * /*v.*/y);

    return /*v.*/length;
}
```

- Here `Vector2d` defines a scope of the structure and `::` is the *scope* operator.

How to Implement Methods of a Structure?

- There is no need to provide a name of the current object — it is implied **implicitly**!

int x,

```
void Vector2d::multByScalar(double z)
{
    x *= z;
    y *= z;
}

double Vector2d::calcLength()
{
    // if the value has not been calculated previously...
    if(length < 0)
        length = sqrt(x * x + y * y);

    return length;
}
```

How to Implement Methods of a Structure?

- Now, how to return an object in the method `multByScalarEnh()`?

```
void Vector2d::multByScalar(double z)
{
    x *= z;
    y *= z;
}

double Vector2d::calcLength()
{
    // if the value has not been calculated previously...
    if(length < 0)
        length = sqrt(x * x + y * y);

    return length;
}
```

Vektor2d

```
Vector2d& Vector2d::multByScalarEnh(double z)
{
    x *= z;
    y *= z;

    return (*this);
}
```

- By using the `this` keyword!

this Keyword

- Represents a *pointer*[1] to the current object, which is called instance.
- Can be used when the explicit referencing of the instance is needed.

v1 2 3 0

FF00

this = &v1

```
int main()
{
    Vector2d v1 = {2, 3, 0};
    Vector2d v2 = {3, 4, 0};

    v1.multByScalar(10);

    v2.multByScalar(10);
```

this = &v2

EE10

v2 3 4 0

int x,

```
void Vector2d::multByScalar(double z)
{
    /* this-> */ x *= z;
    /* this-> */ y *= z;
}
```

this->x

[1] a *pointer* is a holder for an address

The this Keyword

- The keyword `this` can be used in an implicit context as well, but it is redundant!
  - unlike Python, where similar `self` keyword is a must.
- **The rule**: never use `this` keyword unless it really becomes necessary!

```
void Vector2d::multByScalar(double z)
{
    x *= z;
    y *= z;
}
```

pretty OK!

```
void Vector2d::multByScalar(double z)
{
    this->x *= z;
    this->y *= z;
}
```

correct, but redundant!

How to Obtain an Object from a Pointer?

Vector2d* this

```
Vector2d& Vector2d::multByScalarEnh(double z)
{
    x *= z;
    y *= z;

    return (*this);
}


Vector2d& Vector2d::multByScalarEnh(double z)
{
    Vector2d& curInsta = *this;

    curInsta.x *= z;
    curInsta.y *= z;

    return curInsta;
}
```

* here is the dereference operator
  - do not mix it with the multiplication operator, which has the same symbol.

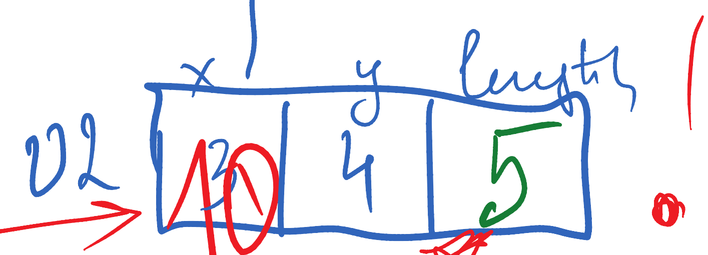
The Problem of Data Inconsistency

```
double Vector2d::calcLength()
{
    // if the value has not been calculated previously...
    if(length < 0)
        length = sqrt(x * x + y * y);

    return length;
}
```

```
int main()
{
    Vector2d v1 = {2, 3, 0};
    Vector2d v2 = {3, 4, 0};
```

⇒

```
int main()
{
    Vector2d v1 = {2, 3, -1};
    Vector2d v2 = {3, 4, -1};
```




```
double l1 = v1.calcLength();
double l2 = v2.calcLength();

v2.x = 10;
double l1_2 = v2.calcLength();        // here l1_2 == 5, which is incorrect
```

The Problem of Data Inconsistency

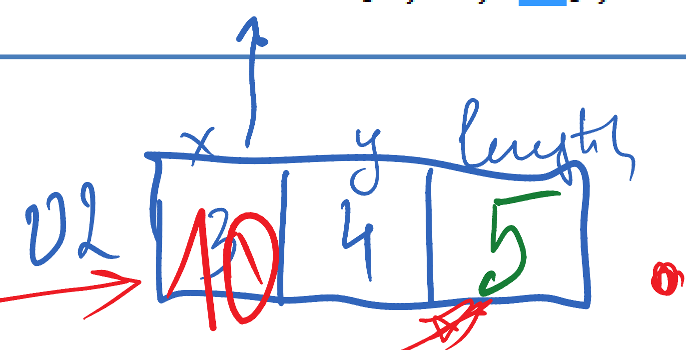
```
double Vector2d::calcLength()
{
    // if the value has not been calculated previously...
```

Two possible solutions:

1) prohibit changing x and y;
2) changing x or y must invalidate the value of `length`.

```
int main(
{
    Vector2d v1 = {2, 3, 0};
    Vector2d v2 = {3, 4, 0};
```

```
{
    Vector2d v1 = {2, 3, -1};
    Vector2d v2 = {3, 4, -1};
```




```
double l1 = v1.calcLength();
double l2 = v2.calcLength();

v2.x = 10;
double l1_2 = v2.calcLength();        // here l1_2 == 5, which is incorrect
```

Object-oriented approach

ENCAPSULATION

Make all Fields Inaccessible from the Outside of the Structure

Step 1: Add *Class Access Modifiers*

Step 2: Put public part of the class (interface) to the top of the declaration

```
struct Vector2d
{
private:
    //----< Fields >----

    double x;
    double y;
    double length;

public:
    //----< Methods >----

    void multByScalar(double z);
    Vector2d& multByScalarEnh(double z);
    double calcLength();
}; // struct Vector2d
```

implement part

interface part

private
int z;

Vector2d v;

x y length z

10 | 1.5 | -1 | 15

```
struct Vector2d
{
public:
    //----< Methods >----

    void multByScalar(double z);
    Vector2d& multByScalarEnh(double z);
    double calcLength();

private:
    //----< Fields >----

    double x;
    double y;
    double length;
}; // struct Vector2d
```

interface

m_

Make all Fields Inaccessible from the Outside of the Structure

Step 2: Put public part of the class (interface) to the top of the declaration

```
struct Vector2d
{
public:
    //----< Methods >----

    void multByScalar(double z);
    Vector2d& multByScalarEnh(double z);
    double calcLength();

private:
    //----< Fields >----

    double x;
    double y;
    double length;
}; // struct Vector2d
```

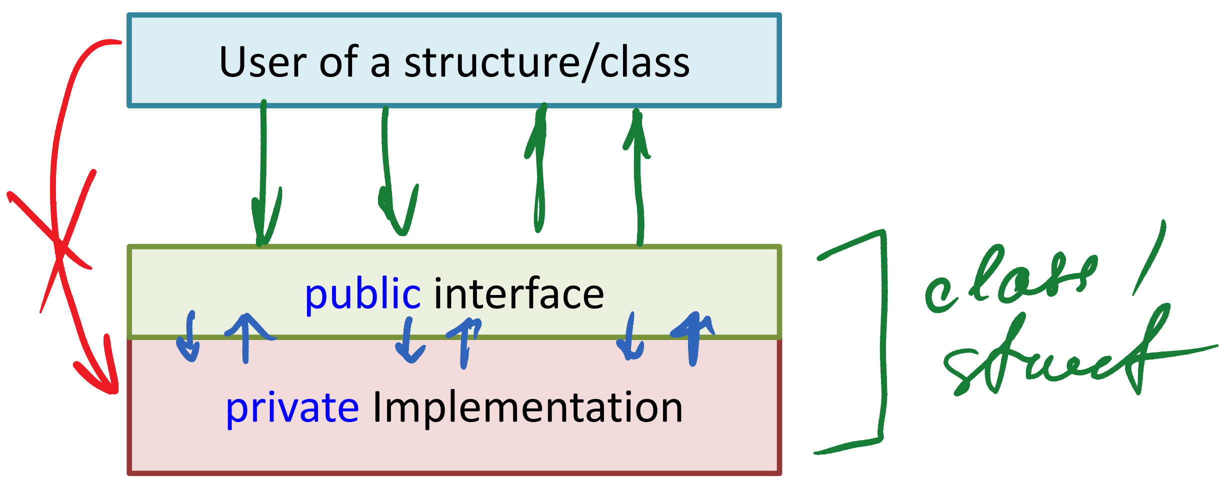
Step 3: According to the *Code Style Rules*, all non-public fields are named with _

```
struct Vector2d
{
public:
    //----< Methods >----

    void multByScalar(double z);
    Vector2d& multByScalarEnh(double z);
    double calcLength();

private:
    //----< Fields >----

    double _x;
    double _y;
    double _length;
}; // struct Vector2d
```

Access Control: *Class Access Modifiers*

- The access to the members of a structure (or class) is controlled by using *Class Access Modifiers*:
  - `private` identifies structure/class members that are only directly accessibly inside a structure/class;
    - serves as a structure/class *implementation* part;
  - `public` identifies structure/class members that are accessible from both inside and outside of the structure/class;
    - such members constitute the public *interface* for a structure/class (its abstraction);

- The public members of a structure/class act as an intermediary between a program and the structure/class private members.



Encapsulation and Data Hiding

- *Encapsulation* is gathering the implementation details together and separating them from the abstraction.

- *Data hiding* (putting data into the private section of a class) is an *instance of encapsulation*, and so is hiding functional details of an implementation in the private section.

public interface:
public *methods* (functions)

private implementation:
private *fields* and *methods*

and (very rarely) public *fields* (variables)

How to Initialize a Structure Now?

{ _length = -1 }

```
struct Vector2d
{
public:
    //----< Methods >----
    void init();
    void multByScalar(double z);
    Vector2d& multByScalarEnh(double z);
    double calcLength();

private:
    //----< Fields >----

    double _x;
    double _y;
    double _length;
}; // struct Vector2d
```

```
int main()
{
    Vector2d v1 = {2, 3, -1};
    Vector2d v2 = {3, 4, -1};
    v1.init();
```

Fields are not accessible anymore!

We need to create a special public (*interface*) method which makes all the work for us!

Initialize the Structure by Using a Constructor Method

specific
the instance
of the structure

Vector2d* this

```
struct Vector2d
{
public:
    //----< Methods >----

    Vector2d();
    Vector2d(double x, double y);

    void multByScalar(double z);
    Vector2d& multByScalarEnh(double z);
    double calcLength();

private:
    //----< Fields >----

    double _x;
    double _y;
    double _length;
}; // struct Vector2d
```

```
Vector2d::Vector2d()
{
    _x = 0;
    _y = 0;
    _length = -1;
}
```

Vector2d* this

```
Vector2d::Vector2d(double x, double y)
{
    _x = x;
    _y = y;
    _length = -1;
}
```

this->

Vector2d v;
the default constructor

Structure/Class Constructor

- A *constructor* is a special method (function) of a class that is called automatically when an object of the class is being created;
  - has exactly the same name as the class;
    - for a class `Foo` its constructor is `Foo::Foo();`
  - can have different parameters:
    - the constructor with no parameters is the *default constructor*: `Foo::Foo();`
    - a constructor with arbitrary parameters is one of the possible initialization constructors: `Foo::Foo(int a);`
    - there are also a few constructors with special meanings: the *copy constructor*, the *move constructor*;
  - has no return value:
    - `Foo::Foo() { }`
    - ~~`Foo`~~ `Foo::Foo() { }`
    - ~~`void`~~ `Foo::Foo() { }`
    - ~~`int`~~ `Foo::Foo() { }`

The Member Initializer List

- The *member initializer list* consists of a comma-separated list of initializers preceded by a colon.
- Must be used in order to *initialize* member fields instead of *re-assigning* their values:

cout << _x;

the header

```
Vector2d::Vector2d()
{
    _x = 0;
    _y = 0;
    _length = -1;
}
```

Mind the neck, boy!

```
Vector2d::Vector2d()
    : _x(0) , _y(0) , _length(-1)
{
    // _x = 0;
    // _y = 0;
    // _length = -1;
}
```

the body

```
Vector2d::Vector2d(double x, doubl
{
    _x = x;
    _y = y;
    _length = -1;
}
```

Aaah! This is why he asks you putting the opening bracket to a new line!

```
Vector2d::Vector2d(double x, double y)
    : _x(x)
    , _y(y)
    , _length(-1)
{
    // _x = x;
    // _y = y;
    // _length = -1;
}
```

_x(x ≠ x)

What Is the Difference Between the Structures and the Classes?

Structure

- is a custom datatype
- declared with `struct` keyword
- all members are `public` by default

Class

- is a custom datatype
- declared with `class` keyword
- all members are `private` by default

no. more. difference.