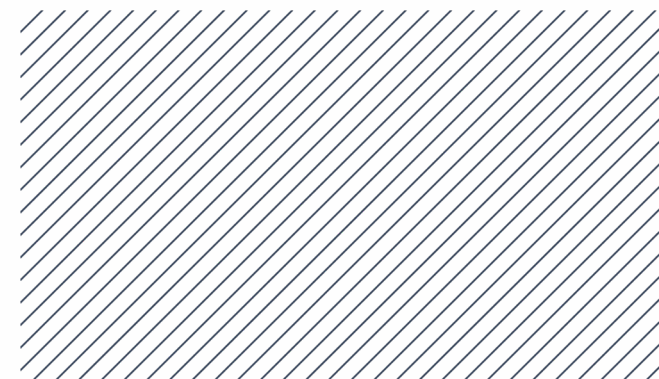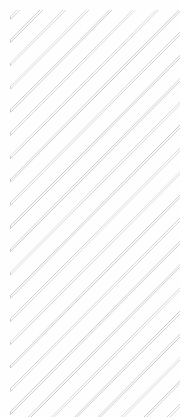# Lecture 9

Classes and structures, pt.1

Konstantin Leladze

C++ Basics

DIHT MIPT 2021

Structures, motivation

Let's have a look on the solution of the problem about perimeter of convex polygon, which we have discussed in the previous lesson.

Problem:
- We have to use two different arrays for storing x and y coordinate.
- If we formulate this problem in such a way that it needs to be solved in 3D, then the number of arrays will increase to 3.
- Maybe there is a tool thanks to which we could declare our own user-defined type and store these three variables of type double inside one single type?

```cpp
#include <iostream>
#include <limits>
#include <cmath>


double dot_product (
    const double x1,
    const double y1,
    const double x2,
    const double y2
) {
    return x1 * x2 + y1 * y2;
}

int main() {
    int n;
    std::cin >> n;
    double* const x = new double[n];
    double* const y = new double[n];

    for (int i = 0; i < n; i ++)
        std::cin >> x[i] >> y[i];

    int perimeter = 0;

    if (n >= 3)
        for (int i = 0; i < n; i ++) {
            const int j = (i + 1) % n;
            const int v_x = x[j] - x[i];
            const int v_y = y[j] - y[i];
            perimeter += dot_product(v_x, v_y, v_x, v_y);
        }

    std::cout << perimeter << std::endl;

    delete[] x;
    delete[] y;
    return 0;
}
```

Structures, definition

The struct keyword allows us to define our own custom type, which can (and in our case will) contain variables.

The code shown in the screenshot on the right only defines the coordinates type, **but does not create a variable of its type**.

In-class declared variable is called a member field.

For example, in this struct we have 2 member fields: **x** and **y**.

```
struct coordinates {
    double x;
    double y;
}
```

Instance VS Definition

To create a variable of our own specified type **coordinates**, we need to write the following code (see int main). Now, variable **c** is called instance of structure coordinates.

The number of instances of the coordinates structure is not limited. You can imagine a struct is a blueprint and its instances are products made from this blueprint.

The variables **x** and **y** are contained within the coordinates type. To access them (both read and write), you need to use a special **operator .** (will be discussed in the future).

Their lifetime is limited by the lifetime of an instance of structure coordinates: when it is destroyed, so are the variables it contains.

```cpp
#include <iostream>

struct coordinates {
    double x;
    double y;
};

int main () {
    coordinates c;
    c.x = 123.3;
    c.y = 32.5;
}
```

Constructor

Let's imagine that we have created a structure containing 10 internal types. Agree, writing 10 lines of code to initialize each variable inside is not the most pleasant job.

There is such a thing as a constructor. The constructor allows you to initialize all variables inside the structure instance with values. On the right, I gave an example of the **direct-list-initialization** constructor.

```cpp
#include <iostream>

struct coordinates {
    double x;
    double y;
};


int main () {
    coordinates c({ 123.3, 32.5 });
    std::cout << c.x << " " << c.y << std::endl;
}
```

Default Constructor

In most cases (in which specifically we will discuss later), the structure has a **default constructor** which does not accept any arguments and initializes all fields by default.

For example, in the case of our structure, the default constructor will fill **x** and **y** with zeros (and not random values, how would it happen if we set these variables in the main function)

```cpp
#include <iostream>

struct coordinates {
    double x;
    double y;
};


int main () {
    coordinates c;
    std::cout << c.x << " " << c.y << std::endl;
}
```

Methods

In addition to variables, we can define functions inside structures. A function defined inside structures is called a method.

In this case, all the variables contained in the instance of the structure will be available in the body of the function.

To call a method, we need to apply an **operator .** to some instance

```cpp
#include <iostream>
#include <cmath>


double dot_product (
    const double x1,
    const double y1,
    const double x2,
    const double y2
) {
    return x1 * x2 + y1 * y2;
}

struct coordinates {
    double x;
    double y;

    double distance_from_zero () {
        return std::sqrt(dot_product(x, x, y, y));
    }
};


int main () {
    coordinates c({ 2.2, 4.2 });
    std::cout << c.distance_from_zero() << std::endl;
}
```

Static methods, motivation

In the previous example, we defined the **dot_product** function outside of the class. This is not very convenient, because in this case we need to keep in mind the fact that the **coordinates** struct is associated with the **dot_product** function. For example, if we want to transfer the definition of the class coordinates, we will need to transfer the definition of the **dot_product** function (or make a forward declaration)

```cpp
#include <iostream>
#include <cmath>

double dot_product (
    const double x1,
    const double y1,
    const double x2,
    const double y2
);

struct coordinates {
    double x;
    double y;

    double distance_from_zero () {
        return std::sqrt(dot_product(x, x, y, y));
    }
};

double dot_product (
    const double x1,
    const double y1,
    const double x2,
    const double y2
) {
    return x1 * x2 + y1 * y2;
}
```

Static methods

To solve this problem, we can use a **static** method. The Static method is not associated with a specific instance of the structure, but with its type.

Because of this, inside a static method, we cannot access the variables defined inside the class, which, by the way, are called "fields".

You can call a static method from within a class simply by specifying its name.

To call it from outside, you need to apply the **operator ::** to the **type** of the struct (**not to its instance!**)

```cpp
#include <iostream>
#include <cmath>

struct coordinates {
    double x;
    double y;

    double distance_from_zero () {
        return std::sqrt(dot_product(x, x, y, y));
    }

    static double dot_product (
        const double x1,
        const double y1,
        const double x2,
        const double y2
    ) {
        return x1 * x2 + y1 * y2;
    }
};


int main () {
    coordinates c({ 2.2, 4.2 });
    /// Still workds fine:
    std::cout << c.distance_from_zero() << std::endl;
    /// Calling static method
    std::cout << coordinates::dot_product(2.4, 3.2, 5.2, 1.2) << std::endl;
}
```

Const qualified instance

Struct instances can be const-qualified. In this case, we will not be able to change the internal variables of these structures and call non-const methods.

But what exactly is a non-const method? Non-const method is a method that is not defined as const!

```cpp
struct coordinates {
    double x;
    double y;

    double distance_from_zero () {
        return std::sqrt(dot_product(x, x, y, y));
    }

    static double dot_product (
        const double x1,
        const double y1,
        const double x2,
        const double y2
    ) {
        return x1 * x2 + y1 * y2;
    }
};


int main () {
    const coordinates c({ 2.2, 4.2 });
    c.x = 123.3; /// Error.
    std::cout << c.distance_from_zero() << std::endl; /// Error.
}
```

Const Methods

A const method is a
method that can be called
on a const-qualified
instance, but it can NOT
modify the internal fields
of a structure:

```
double instance_from_zero () const {
    x = 123.3; /// CE
    return std::sqrt(x * x + y * y);
}
```

```
main.cpp:17:11: error: assignment of member 'coordinates::x' in read-only object
   17 |          x = 123.3; /// CE
      |          ~~^~~~~~~
```

```cpp
struct coordinates {
    double x;
    double y;

    double distance_from_zero () const {
        return std::sqrt(dot_product(x, x, y, y));
    }

    static double dot_product (
        const double x1,
        const double y1,
        const double x2,
        const double y2
    ) {
        return x1 * x2 + y1 * y2;
    }
};



int main () {
    const coordinates c({ 2.2, 4.2 });
    c.x = 123.3; /// Error.
    std::cout << c.distance_from_zero() << std::endl; /// OK.
}
```

Structures in std

C ++ cannot be imagined without the ability to define your own custom types. In fact, the language itself already has many built-in structure types.

For example:

std::vector
std::cin
std::cout
std::array
std::string
std::allocator
and many, many more…

In fact, most of these types are defined using the class keyword. To understand what its meaning is, let's look at such a thing as encapsulation.

Encapsulation

In object-oriented programming (OOP), **encapsulation** refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components.

Encapsulation is used to hide the values or state of a structured data object inside a class, preventing direct access to them by clients in a way that could expose hidden implementation details or violate state invariance maintained by the methods.

```cpp
#include <iostream>
#include <cmath>

struct a_great_structure {
    int val;
};

int main () {
    a_great_structure s;

    /// Absence of encapsulation (this code works fine)
    s.val = 123;
}
```

Private and public keywords

The special word private allows us to hide the fields of the structure from an external user, prohibit access to them.

```cpp
#include <iostream>
#include <cmath>

struct a_great_structure {
private:
    int val;
};

int main () {
    a_great_structure s;

    /// Presence of encapsulation (this code does NOT work, CE)
    s.val = 123;
}
```

```
main.cpp:21:7: error: 'int a_great_structure::val' is private within this context
   21 |     s.val = 123;
      |       ^~~
main.cpp:14:9: note: declared private here
   14 |     int val;
      |         ^~~
```

The public keyword is the opposite of private. By default, all fields of the structure are set to public.

Sections

When we use a **private** or **public** keyword, we create the corresponding section in the class or struct.

All fields and methods specified after the beginning of a section *(after specifying the word **private** or **public**)*, but before the beginning of the next section, refer to it.

In the screenshot, **private** variables are boxed in **blue**, **public** - in **red**.

```
struct a_s {
    int a;
    float b;
private:
    int c;
    double d;
public:
    char e;
private:
    float f;
private:
    float g;
};
```

struct vs class

There are 2 differences in total between class and struct. These differences are:

1)
- By default, section of **struct** is **public**
- By default, section of **class** is **private**

2) We will discuss the second difference later.

Since there is little difference between class and structure, you can use both words. However, it is common practice to use the word struct only when you are not specifying private fields (not using encapsulation). If your type uses encapsulation, it is better to define it as a class.

```cpp
#include <iostream>
#include <cmath>

struct s_coordinates {
    double x;
    double y;
};

class c_coordinates {
    double x;
    double y;
};


int main () {
    s_coordinates s;
    c_coordinates c;

    s.x = 123;

    /// Error: by default, x is private
    c.x = 123;
}
```

Lecture 9

Classes and structures, pt.1

Konstantin Leladze

C++ Basics

DIHT MIPT 2021