

Енкапсулация. Абстракция. Работа с потоци.

д-р Тодор Цонков: todort@uni-sofia.bg

От какво ще се състои настоящата лекция?

Енкапсулация. Модификатори за достъп. Абстракция. Константни класове и член-функции. Mutable. Потоци и четене и писане от файл.

`std::string` и `std::print`

Какво е енкапсулацията?

- Скриване на вътрешното състояние (data hiding).
- Предоставяне на контролирани методи за достъп (get/set).
- Предпазва от неконсистентни промени и гарантира запазване на инвариантите (условията, при които един клас е валиден).
- Съществуват public, protected и private полета. В момента не сме учили наследяване и private и protected без наследяване се държат еднакво и затова мислим само за private и public.
- Енкапсулацията повишава надеждността, четимостта и поддръжката на кода, тъй като промените във вътрешната реализация не засягат външния интерфейс.
- Възможно е освен полета да има и private член функции на класа.

Пример за енкапсулация

```

#include <iostream>

class Student
{
private:
    int age;
    double grade;

public:
    void print() const
    {
        std::cout << "Age: " << age
                    << ", Grade: " << grade
                    << '\n';
    }
};

```

```

int main()
{
    Student s;

    s.age = 20;      // ✗ error: private member
    s.grade = 5.5;   // ✗ error: private member

    s.print();
}

```

Пример за get/set функции и защо са полезни?

Методи (member functions), КОИТО:
Getter → връща стойността на
частно (private) поле
Setter → променя стойността на
частно поле.

В примера - имаме клас, който
вътрешно поддържа
температура в градуси по
Целзий, а предоставя на
потребителя достъп до
температура по Фаренхайт и
Целзий и няма значение как
вътрешно е имплементирана.

```
class Temperature {  
public:  
  
    // Getter/Setter в Целзий  
    double getCelsius() { return celsius_; }  
    void setCelsius(double c) { celsius_ = c; }  
  
    // Getter/Setter във Фаренхайт (директно конвертира)  
    double getFahrenheit() { return celsius_ * 9.0 / 5.0 + 32.0; }  
    void setFahrenheit(double f) { celsius_ = (f - 32.0) * 5.0 / 9.0; }  
  
private:  
    double celsius_;  
};
```

Пример за енкапсулация с член функции

```
#include <iostream>

class Student
{
private:
    int age{};
    double gradeValue{};

    void validate() const
    {
        if (age < 0)
        {
            std::cout << "Invalid age\n";
        }
    }

    void printInternal() const
    {
        std::cout << "Age: " << age
                    << ", Grade: " << gradeValue
                    << '\n';
    }
}
```

```
public:
    void print() const
    {
        validate();           // достъпно вътре в класа
        printInternal();      // достъпно вътре в класа
    }
};

int main()
{
    Student s;               // създаваме обект от класа, който ще ползваме
    s.print();               // OK

    return 0;
}
```

Сравнение с языка C#

```
using System;

class Temperature
{
    private double celsius_;

    // Конструктор
    public Temperature(double celsius)
    {
        celsius_ = celsius;
    }

    // Свойство за Целзий
    public double Celsius
    {
        get { return celsius_; }
        set { celsius_ = value; }
    }

    // Свойство за Фаренхайд (директно изчислява/променя)
    public double Fahrenheit
    {
        get { return celsius_ * 9.0 / 5.0 + 32.0; }
        set { celsius_ = (value - 32.0) * 5.0 / 9.0; }
    }
}
```

```
class Program
{
    static void Main()
    {
        Temperature t = new Temperature(25.0);

        Console.WriteLine($"Celsius: {t.Celsius}°C");           // 25°C
        Console.WriteLine($"Fahrenheit: {t.Fahrenheit}°F");     // 77°F

        t.Fahrenheit = 212.0;

        Console.WriteLine($"Celsius: {t.Celsius}°C");           // 100°C
        Console.WriteLine($"Fahrenheit: {t.Fahrenheit}°F");     // 212°F
    }
}
```

Какво е абстракцията?

- 1) Скриване на сложността → по-лесен и безопасен код и подобрена четимост и поддръжка
- 2) Позволява работа с високо ниво на концепции
- 3) Улеснява повторна употреба на код
- 4) Позволява да се работи с обекти на ниво какво правят, а не как точно го правят.
- 5) Помага за по-добра модулност и разделяне на отговорностите, тъй като всяка част от системата гледа само своя „абстрактен“ интерфейс.

```
class Car
{
private:
    int speed;
    double fuel;

    // refuel е вътрешна, private функция
    void refuel(double amount)
    {
        fuel += amount;
        std::cout << "Refueled internally " << amount << " liters.\n";
    }

public:
    void drive(int distance)
    {
        // ако горивото е твърде малко, Car автоматично се презарежда
        if(fuel < distance * 0.1)
        {
            refuel(20); // private refuel се извиква вътре
        }

        std::cout << "Car drives " << distance << " km.\n";
        speed = 60;
        fuel -= distance * 0.1;
    }

    void printStatus() const
    {
        std::cout << "Speed: " << speed << " km/h, Fuel: " << fuel << "
        liters\n";
    }
};
```

Каква е разликата между енкапсулация и абстракция?

1) При абстракцията се крият от потребителят несъществените детайли на обекта - т.е. как го прави, а се показва какво прави, докато енкапсулацията крие член-променливите на класа.

2) Абстракцията се постига чрез използване на публичните методи на класа, а енкапсулацията чрез слагане на private методи и полета на класа.

3) Начин на мислене:

Енкапсулация = „заклучваме вратата, за да не могат да променят вътрешността директно“.

Абстракция = „показваме само бутона за стартиране, без да обясняваме двигателя под капака“.

Константността в езика C++

Константността в C++ е механизъм за гарантиране, че определени данни няма да бъдат променяни. Защо е полезна:

- 1)Предотвратява случайна модификация на данните.
- 2)Подобрява разбирането на кода.
- 3)Позволява оптимизации от компилатора.
- 4)Безопасност при четенето на данни

Константността при класовете -

- 1) При класове може да бъдат константни член функциите - Всеки метод, който не променя състоянието на обекта, трябва да има `const` накрая.
- 2) Също така може да имаме константни обекти от класове, които могат да извикват само `const` функции
- 3) При тях трябва да се създаде само веднъж обекта и полетата обикновено да са `private`.

Пример на C++ и Rust

```
class Point
{
private:
    int x, y;

public:
    int getX() const { return x; }
    int getY() const { return y; }

    void setX(int value) { x = value; }
    void setY(int value) { y = value; }
};

int main()
{
    const Point p; // константен обект
    p.getX();      // ✅ ок
    p.setX(5);     // ❌ грешка! const обект не може да се модифицира
}
```

```
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn new(x: f64, y: f64) -> Self {
        Point { x, y }
    }

    fn print(&self) {
        println!("Point({}, {})", self.x, self.y);
    }
}

fn main() {
    let p = Point::new(3.0, 4.5);
    p.print();
}
```

Константността като дизайн в C++

`int getX() const;`
означава, че тази функция гарантира, че няма да промени състоянието на обекта. Потребителят на класа може да разчита на това и е част от публичния интерфейс, който да ползва потребителят

```
void process(const std::vector<int>& data)
{
    data.push_back(5); // ❌
}
```

компилационна грешка

1) Добро правило - всичко да е `const`, ако не променя (като на Rust).

2) `const Car&` - по-добре от документация показва, че класът няма да се променя и копира.

3) Ако обектът не трябва да се променя и се сложи `const`, то той ще се държи като примитивен тип (`int`, `double`) и по този начин ще може да се ползва като стойност - value semantics.

Ключова дума mutable в C++


Позволява модификация на членове въпреки const функцията.

Полезна е за:

Кеширане на резултати, Броене на достъпи, Логове

Пример:

```
class Counter {  
    mutable int count = 0;  
public:  
    void increment() const { count++; }  
    int getCount() const { return count; }  
};
```

```
int main(){  
    const Counter c;  
    c.increment(); //  работи,  
    защото count е mutable  
    std::cout << c.getCount();  
    return 0;  
}
```

Без mutable компилаторът би върнал грешка, че не може да се променя елемент в константа функция.

Позволява да се пазят вътрешни данни, които не променят семантиката на обекта.

Потоци. Четене и писане от файл.

Потоците са абстракция за вход и изход (I/O) на данни и са последователност от данни, която може да бъде прочетена или записана.

Позволяват работа с файлове, клавиатура и други устройства.

Намират се:

```
#include <iostream> // std::cin,  
std::cout, std::cerr - std::cerr се  
ползва за записване на данни в  
буфера за грешки.
```

```
#include <fstream> // std::ifstream,  
std::ofstream, std::fstream
```

std::ifstream се ползва за четене от файл.

```
std::ifstream file("data.txt");
```

std::ofstream се ползва за писане във файл: std::ofstream file("output.txt");
std::fstream - за четене и писане:

```
std::fstream file("data.txt", std::ios::in |  
std::ios::out);
```

Отворили сме го едновременно за четене и писане.

Потоци. Продължение.

Режимите на отваряне са:

<code>std::ios::in</code>	Четене
<code>std::ios::out</code>	Писане
<code>std::ios::app</code>	Добавяне в края
<code>std::ios::ate</code>	Отиване в края при отваряне
<code>std::ios::trunc</code>	Изтрива съдържанието
<code>std::ios::binary</code>	Бинарен режим

Може да проверим дали файлът е отворен успешно така:

```
if (!file.is_open())
{
    std::cout << "Cannot open file\n";
}
```

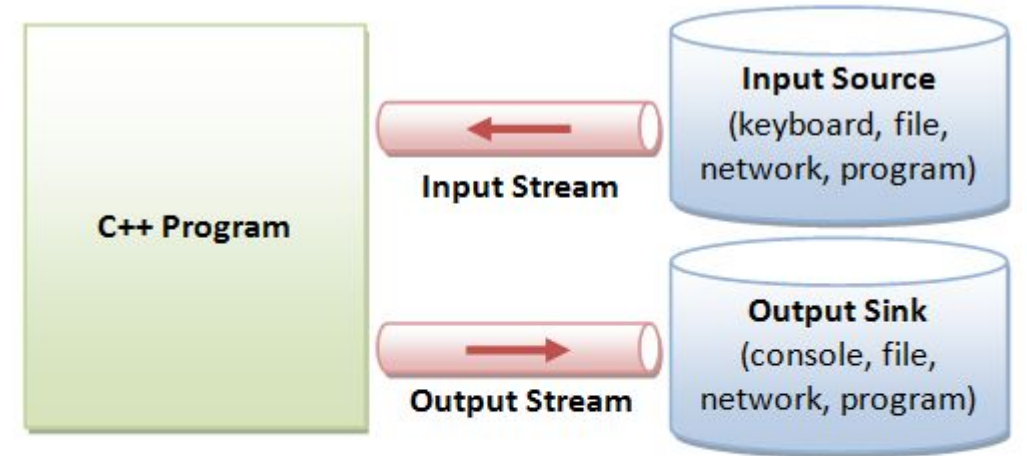
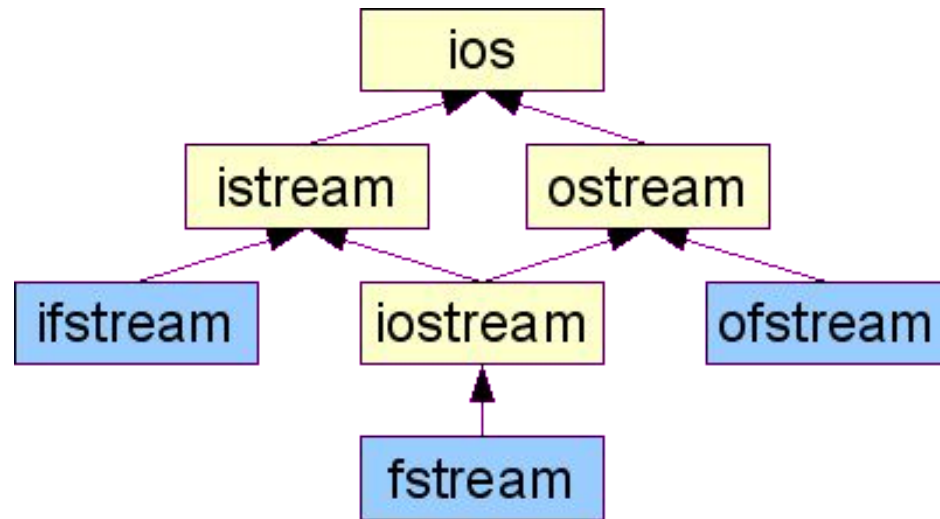
Четенето от файла става като `cin`:

```
std::ifstream file("data.txt");
int x;
file >> x; // спира на whitespace
std::string line;
while (std::getline(file, line))
{
    std::cout << line << "\n";
} // спира на нов ред
```

Писането като `cout`:

```
std::ofstream file("output.txt");
file << "Hello\n";
file << 42;
```

Вход/Исход на данни



Internal Data Formats:

- Text: char, wchar_t
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

Потоци. Примери и добри практики.

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ofstream out("numbers.txt");
    out << 10 << " " << 20 << " " << endl;
    out.close();
    ifstream in("numbers.txt");
    int a, b;
    in >> a >> b;
    cout << a << ", " << b << endl;
    in.close();
    return 0;
}
```

Добри практики:

- 1) Винаги проверявайте дали файлът е успешно отворен.
- 2) Затваряйте файловете след работа (close() или автоматично чрез fstream).
- 3) Използвайте буфери и flush() при нужда от незабавен запис.
- 4) Избягвайте хардкоднати пътища – използвайте относителни или конфигурационни файлове.
- 5) Потоците са универсален инструмент за работа с вход и изход в C++.

Какво е std::string - стандартът в C++ за работа с

ТЕКСТ

std::string е стандартният клас за работа с текст (низове) в C++.

Той представлява динамичен контейнер от символи, който автоматично управлява паметта си. Намира се в <string>

Може да расте и намалява и се намира на непрекъснатата последователност от памет(C++ 11).

Обикновено съдържа вътрешно:
char* data; size_t size; size_t capacity;

Основни операции:

- 1) Конструирание:
std::string s1 = "Hello";
std::string s2(5, 'a'); // "aaaaa"
- 2) Размер - s.size() ил s.length(); // същото
- 3) Добавяне:
s += " text";
s.append(" more");
- 4) Вмъкване:
s.insert(0, "Start ");
- 5) Изтриване:
s.erase(0, 5);
- 6) Търсене:
s.find("lo")
- 7) Подстринг:
s.substr(0, 3);

Модерен начин за печатане на данни.

`std::format` (C++20) форматира данни в низ (`std::string`), използвайки placeholders `{}` или индекси `{0}` `{1}` и правила за форматиране.

Използва се с: `#include <format>`

```
std::format("{:.2f}", 3.14159); // 3.14
std::format("{:08}", 42);       // 00000042
std::format("{:x}", 255);       // ff
std::format("{:#x}", 255);      // 0xff
```

```
std::format("|{:>10}|", 42); // десен align
std::format("|{:<10}|", 42); // ляв
std::format("|{: ^10}|", 42); // център
```

`std::print` форматира данни и ги извежда директно към стандартния изход (`stdout`) или `stderr` или файл по Ваш избор. Подобно е на печатането в езика Python.

Използва се с `<include print>`

Работи на принципа на `std::format`

Примери:

```
int a = 3, b = 7;
```

```
1)std::cout << "a = " << a << ", b = " << b << '\n';
```

```
2)std::println("a = {}, b = {}", a, b);
```

```
3)std::println("|{:>10}|", 42); // десен align
```

4) Файл по Ваш избор:

```
std::ofstream file("out.txt");
std::print(file, "Hello {}\n", 123);
}
```

Сравнение между `std::cout` и `std::print`

`std::print` е по-бърз от `std::cout`, защото не използва `iostream`, който е силно йерархичен и много абстрация, не флъшва `streams`.

При реални тестове `std::print` е 2-3 пъти по-бърз от `std::cout`.

При много аргументи е много по-четим:

```
std::cout << "User " << name << " age " <<
age << " score " << score << "\n";
```

```
std::println("User {} age {} score {}",
             name, age, score);
```

`std::cout` е:

- 1) Исторически стандарт
- 2) Тежък
- 3) Verbose
- 4) Подходящ за legacy

`std::print` е:

- 1) Модерен
- 2) По-бърз
- 3) По-четим
- 4) По-близо до реалните нужди

При нов проект започвайте с `std::print`, при стара база от код не го пренаписвайте, а ползвайте `std::cout` освен, ако записването е много бавно.

Въпроси?

Благодаря за вниманието!

допълнителни материали: learncpp.com, глава 28