Енкапсулация. Абстракция. Работа с потоци.

титуляр на курса: д-р Тодор Цонков
(ttsonkov@gmail.com)

От какво ще се състои настоящата лекция?

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

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

```cpp
#include <iostream>

class Student
{
private:
    int age;
    double grade;

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

```cpp
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

```cpp
#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';
    }
```

```cpp
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;
}
```

Защо енкапсулацията е важна?

```
class Temperature {
private:
    double celsius_; // internal representation
public:
    // Setter: задава температура във Фаренхайт
    void setFahrenheit(double f) {
        celsius_ = (f - 32.0) * 5.0 / 9.0;
    }
    // Getter: връща температура във Фаренхайт
    double getFahrenheit() const {
        return celsius_ * 9.0 / 5.0 + 32.0;
    }
    // Опционално: getter за Celsius
    double getCelsius() const {
        return celsius_;
    }
};
```

```
#include <iostream>
#include <cmath>
class Point {
private:
    double r_ = 0.0;    // радиус
    double phi_ = 0.0;  // ъгъл в радиани
public:
    // Конструктор чрез декартови координати
    Point(double x = 0.0, double y = 0.0) {
        setXY(x, y);
    }
    // Setter: задава координати чрез x, y
    void setXY(double x, double y) {
        r_ = std::sqrt(x * x + y * y);
        phi_ = std::atan2(y, x);
    }
    // Getter: връща x координата
    double x() const {
        return r_ * std::cos(phi_);
    }
    // Getter: връща y координата
    double y() const {
        return r_ * std::sin(phi_);
    }
    // Принтира координатите
    void print() const {
        std::cout << "(" << x() << ", " << y() <<
")\n};
};
```

Сравнение с езика Java

C++

```cpp
class Temperature {
private:
    double celsius_; // internal representation
public:
    // Setter: задава температура във Фаренхайт
    void setFahrenheit(double f) {
        celsius_ = (f - 32.0) * 5.0 / 9.0;
    }
    // Getter: връща температура във Фаренхайт
    double getFahrenheit() const {
        return celsius_ * 9.0 / 5.0 + 32.0;
    }
    // Опционално: getter за Celsius
    double getCelsius() const {
        return celsius_;
    }
};
```

Java

```java
public class Temperature {
    // internal representation
    private double celsius;

    // Setter: задава температура във Фаренхайт
    public void setFahrenheit(double f) {
        this.celsius = (f - 32.0) * 5.0 / 9.0;
    }

    // Getter: връща температура във Фаренхайт
    public double getFahrenheit() {
        return this.celsius * 9.0 / 5.0 + 32.0;
    }

    // Опционално: getter за Celsius
    public double getCelsius() {
        return this.celsius;
    }
}
```

Абстракция

- Скриване на сложността → по-лесен и безопасен код

- Подобрена четимост и поддръжка
- Позволява работа с високо ниво на концепции

- Улеснява повторна употреба на код

- Позволява да се работи с обекти на ниво какво правят, а не как точно го правят.
- Помага за по-добра модулност и разделяне на отговорностите, тъй като всяка част от системата гледа само своя „абстрактен“ интерфейс.

```
class Car {
public:
    void start() {
        checkEngine();
        std::cout << "Колата стартира\n";
    }
    void stop() {
        std::cout << "Колата спира\n";
    }
    void refuel(int liters) {
        fuel += liters;
        std::cout << "Заредени " << liters << " литра гориво\n";
    }
private:
    int fuel = 0;
    // Детайлът се скрива от потребителя
    void checkEngine() {
        if(fuel <= 0) {
            std::cout << "Грешка: Няма гориво!\n";
        }
    }
};
```

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. Безопастност при четенето на данни
5. По подразбиране всички променливи в C++ могат да бъдат променяни, докато в езици като Rust е обратното

```cpp
//C++
class Point {
    double x, y;
public:
    double getX() const { return x; } // //const function
    double getY() const { return y; }
    void setX(double val) { x = val; }
};
//Rust
struct Point {
    x: f64,
    y: f64,
}

impl Point {
    fn get_x(&self) -> f64 {
        self.x
    }

    fn get_y(&self) -> f64 {
        self.y
    }

    fn set_x(&mut self, val: f64) {
        self.x = val;
    }
}
```

Константни обекти

1. Константен обект може да извиква само константни функции.
2. Позволява работа с обекти, които не трябва да се променят.
3. Защитава обекта от случайни промени.
4. Ясно показва на програмиста кои функции не модифицират състоянието.

```
C++:
const Point p;
// p.setX(5); // ❌ грешка: не може да се извика неконстантна функция
double x = p.getX(); // ✅ позволено

Rust:
    let p = Point { x: 1.0, y: 2.0 };   // immutable променлива (аналог на const Point)
    // p.set_x(5.0);     // ❌ грешка: не може да заемем &mut self от immutable променлива

    let x = p.get_x();   // ✅ позволено (само &self)
    println!("{}", x);

    let mut q = Point { x: 1.0, y: 2.0 }; // mutable променлива
```

Константни класове

1. Предимство: След създаване на обекта, стойностите му не могат да се променят.
2. Подходящ за непроменяеми структури, например координати на точки, цветове, конфигурации.
3. Използва се често при многонишковото програмиране
4. Не съществуват като понятие в езика C++, но могат да бъдат направени с конструкциите на езика

```
class ImmutablePoint {
    double x, y;
public:
    double getX() const { return x; }
    double getY() const { return y; }
};
```

Ключова дума mutable

дефиниция

Позволява модификация на членове въпреки const функцията.

Полезно за:

1)Кеширане на резултати
2)Броене на достъпи
3)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;
}
```

Копие

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.

Обобщение на наученото до момента

```cpp
#include <iostream>
#include <string>

class Student {
private:
    std::string name_;
    int age_;
    mutable int accessCount_ = 0; // mutable за броене на достъпи до
public:
     //Generate constructor!
    // Getter-и (константни) - абстракция и const correctness
    std::string getName() const {
        accessCount_++; // брои всяко извикване
        return name_;
    }
    int getAge() const {
        accessCount_++;
        return age_;
    }
    // Setter-и (енткапсулация)
    void setName(const std::string& name) { name_ = name; }
    void setAge(int age) { age_ = age; }

    // Метод за извеждане (const)
    void print() const {
        accessCount_++;
        std::cout << "Name: " << name_  << ", Age: " << age_
    }
    // Вътрешна статистика (mutable)
    int getAccessCount() const { return accessCount_; }
};
int main() {
    const Student s("Ivan", 20, 5.50);
    s.print();              // ✅ const функция
    std::cout << s.getName() << std::endl; // ✅ const функция
    std::cout << "Access count: " << s.getAccessCount() << std::endl;
 }
```

Потоци. Четене и писане от файл.

Дефиниция

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.

Пример

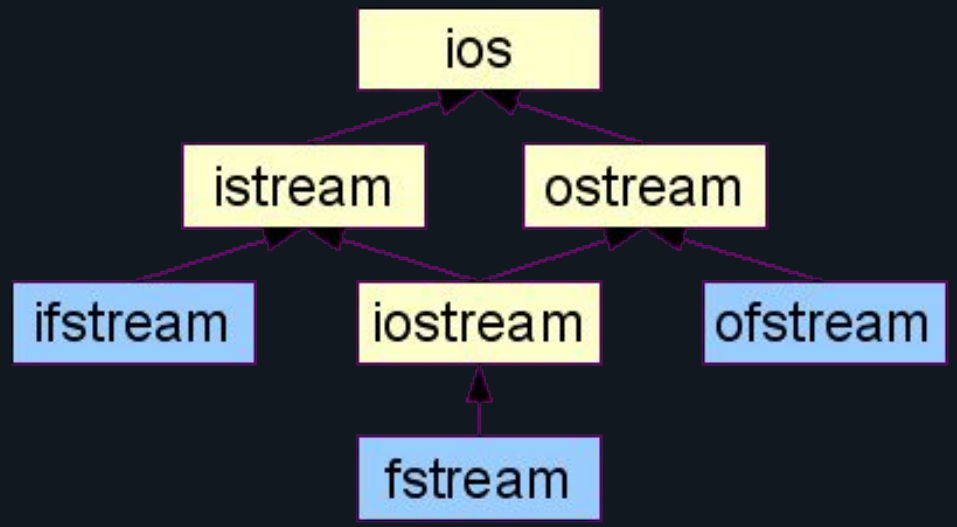
```
#include <iostream>
using namespace std;

int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "You are " << age << "
years old." << endl;
    return 0;
}
```

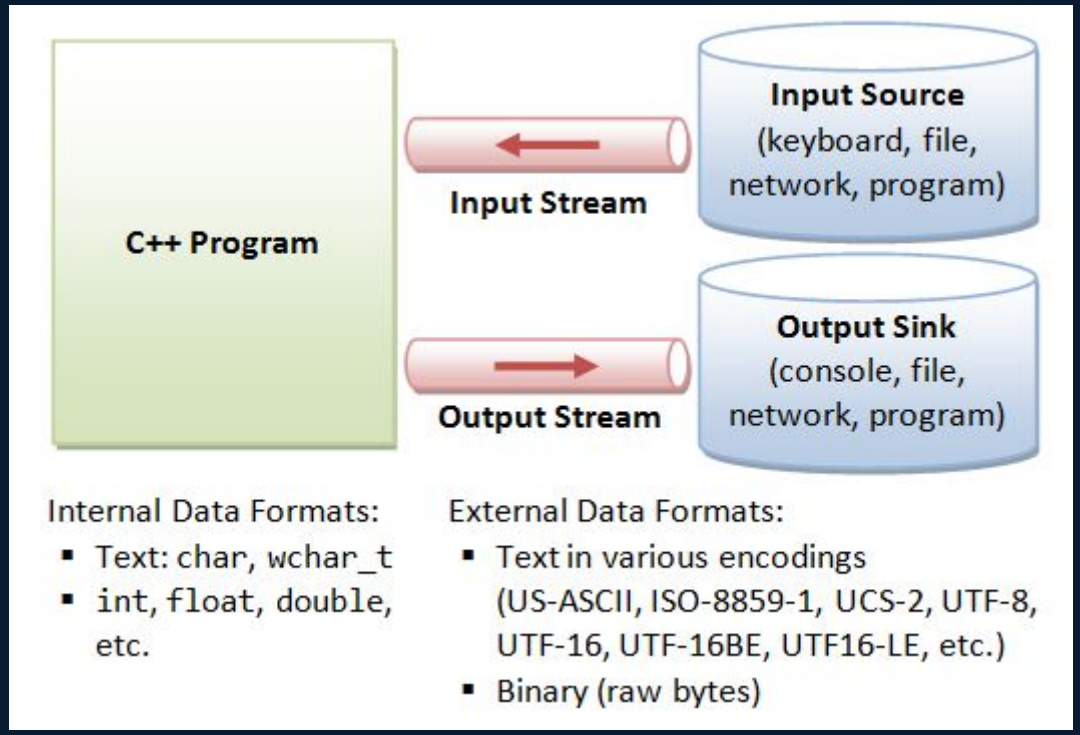
Библиотеки

```
#include <iostream>   // std::cin, std::cout,
std::cerr
#include <fstream>    // std::ifstream,
std::ofstream, std::fstream
#include <sstream>    // std::stringstream,
std::istringstream, std::ostringstream ->
```
текстови потоци

Йерархия на класовете



Визуално обяснение



Потоци - примери

```
#include <iostream>
using namespace std;

int main() {
    string name = "Todor";
    cout << "Hello, " << name << "!" << endl;
    return 0;
}

//endl – нов ред и изчистване на буфера.
//Може да се използва и \n, но endl гарантира
записване в изхода.
```

Пример за четене от файл

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
    ifstream inputFile("data.txt");
    string line;
    if (inputFile.is_open()) {
        while (getline(inputFile, line)) {
            cout << line << endl;
        }
        inputFile.close();
    } else {
        cout << "Unable to open file." <<
endl}
    return 0;
}
```

Потоци - примери

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    fstream file("data.txt", ios::in | ios::out | ios::app);
    file << "Appending this line." << endl;
    file.close();
    return 0;
}

//fstream може едновременно да чете и пише.
//ios::in – четене, ios::out – запис, ios::app – //добавяне в края на файла.
```

Пример за запис във файл

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ofstream outputFile("output.txt");

    if (outputFile.is_open()) {
        outputFile << "Hello, world!" << endl;
        outputFile << "C++ file I/O example." <<
endl;   outputFile.close();
    } else {
        cout << "Unable to open file." << endl;
    }
    return 0;
}
```

Потоци - примери

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ofstream out("numbers.txt");
    out << 10 << " " << 20 << " " << 30 << endl;
    out.close();

    ifstream in("numbers.txt");
    int a, b, c;
    in >> a >> b >> c;
    cout << a << ", " << b << ", " << c << endl;
    in.close();

    return 0;
}

// четене и запис във файл
```

Добри практики:

1)Винаги проверявайте дали файлът е успешно отворен.

2)Затваряйте файловете след работа (close() или RAII 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++.

Модерен начин за печатане на данни

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

std::print форматира данни и ги извежда директно към стандартния изход (stdout) или stderr.
Подобно е на печатенето в езика Python.

Примери:
```
int a = 3, b = 7;
1)std::cout << "a = " << a << ", b = " << b << '\n';

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

Примери:

```cpp
#include <print>
#include <iostream>
#include <string>

int main() {

    // 2️⃣ Използване на placeholders
    int age = 25;
    std::string name = "Ivan";
    std::print("Name: {}, Age: {}\n", name, age);

    // 3️⃣ std::println автоматично добавя нов ред
    int x = 42;
    std::string msg = "Result";
    std::println("{} = {}", msg, x);

    // 4️⃣ Форматиране на числа
    double pi = 3.14159;
    std::println("Pi rounded: {:.2f}", pi); // 3.14
    std::println("Pi rounded: {:.4f}", pi); // 3.1416

    // 6️⃣ Пълнеж с конкретен символ
    std::println("{:*^12}", "C++"); // ****C++****

    // 7️⃣ Булеви и символи
    bool ok = true;
    char c = 'A';
    std::println("ok = {}, c = {}", ok, c);

    // 8️⃣ Извеждане към stderr
    int error_code = 404;
    std::println(std::cerr, "Error code: {}", error_code);

    // 9️⃣ В цикъл
    for (int i = 0; i < 5; ++i) {
        std::println("i = {}", i);
    }

    return 0;
}
```

Въпроси?

Благодаря за вниманието!

допълнителни материали: learncpp.com, глава 28