**1. Структури (struct)**

Структурите в C++ са колекции от свързани данни, които могат да съдържат различни типове. Те са подобни на класовете, но по подразбиране членовете им са публични (public). Използването на struct е полезно, когато искаме да групираме няколко свързани стойности заедно в един обект.

*#include <iostream>*

*struct Point {*

*int x;*

*int y;*

*};*

*int main() {*

*Point p1 = {10, 20};*

*std::cout << "X: " << p1.x << ", Y: " << p1.y << std::endl;*

*return 0;*

*}*

**Разлика между struct и class**

* При struct, членовете са публични по подразбиране.
* При class, членовете са частни (private) по подразбиране.
* struct се използва често за по-прости структури от данни, докато class се използва за по-сложни обекти с енкапсулация.

**2. enum (Изброими типове)**

Изброимите типове (enum) позволяват създаване на символични имена за константи, което прави кода по-четим и разбираем. enum се използва, когато имаме фиксиран набор от възможни стойности.

enum Color {RED, GREEN, BLUE};

*int main() {*

*Color c = GREEN;*

*if (c == GREEN) {*

*std::cout << "Color is GREEN" << std::endl;*

*}*

*return 0;*

*}*

**enum class**

От C++11 насам е въведен enum class, който ограничава обхвата на елементите му и предотвратява конфликт с други идентификатори. Това подобрява четимостта и избягва неочаквани конфликти.

enum class Status {OK, ERROR, UNKNOWN};

int main() {

Status s = Status::OK;

return 0;

}

**3. union (Обединение)**

union позволява съхраняване на различни типове данни в една и съща памет. Само един член може да се използва в даден момент. Това е полезно, когато искаме да използваме една и съща памет за различни цели, като например сериализация или работа с различни типове входни данни.

*#include <iostream>*

*union Data {*

*int i;*

*float f;*

*char str[20];*

*};*

*int main() {*

*Data data;*

*data.i = 10;*

*std::cout << "data.i: " << data.i << std::endl;*

*data.f = 220.5;*

*std::cout << "data.f: " << data.f << std::endl;*

*return 0;*

*}*

**4. typedef и using**

typedef и using позволяват създаване на псевдоними за съществуващи типове. Това подобрява четимостта на кода и улеснява поддръжката му.

*#include <iostream>*

*typedef unsigned int uint;*

*using ulong = unsigned long;*

*int main() {*

*uint a = 100;*

*ulong b = 100000;*

*return 0;*

*}*

**5. namespace**

namespace се използва за групиране на свързани функционалности и за избягване на конфликти на имена. Използването на namespace е полезно при големи проекти с много модули.

*#include <iostream>*

*namespace Math {*

*const double PI = 3.14159;*

*double square(double x) { return x \* x; }*

*}*

*int main() {*

*std::cout << "PI: " << Math::PI << std::endl;*

*std::cout << "Square of 4: " << Math::square(4) << std::endl;*

*return 0;*

*}*

**namespace с using**

Ако често използваме елементи от namespace, можем да избегнем namespace:: чрез using, но трябва да внимаваме, за да не създадем конфликти на имена.

*using namespace Math;*

*int main() {*

*std::cout << square(5) << std::endl;*

*return 0;*

*}*

**Заключение**

* struct е подобен на class, но с публични членове по подразбиране.
* enum и enum class позволяват дефиниране на символични стойности и подобряват четимостта на кода.
* union спестява памет, като използва един и същ адрес за различни типове данни.
* typedef и using се използват за създаване на псевдоними, което улеснява писането на кода.
* namespace помага за организиране на кода и избягване на конфликти на имена, особено в големи проекти.

**Обектно ориентирано програмиране - Седмица 2**

**Тема: Шаблони. Функции от по-висок ред (указатели към функции, std::function, lambda)**

**1. Въведение в шаблоните (Templates)**

**Какво представляват шаблоните?**

Шаблоните в C++ са механизъм за обобщено програмиране, който позволява писането на код, който може да работи с различни типове данни, без да се дублира логиката. Използват се предимно за създаване на **шаблонни функции** и **шаблонни класове**.

**Шаблонни функции (Function Templates)**

Шаблонните функции ни позволяват да дефинираме функция, без да задаваме конкретен тип за нейните аргументи и резултат.

***Пример:***

*#include <iostream>*

*// Дефиниция на шаблонна функция*

*template <typename T>*

*T maxValue(T a, T b) {*

*return (a > b) ? a : b;*

*}*

*int main() {*

*std::cout << maxValue(5, 10) << std::endl; // Извежда: 10*

*std::cout << maxValue(3.14, 2.71) << std::endl; // Извежда: 3.14*

*return 0;*

*}*

**Задача 1:**

Напишете шаблонна функция maxValue, която приема два аргумента от произволен тип и връща по-големия от тях.

**Примерно решение:**

*#include <iostream>*

*template <typename T>*

*T maxValue(T a, T b) {*

*return (a > b) ? a : b;*

*}*

*int main() {*

*std::cout << maxValue(42, 35) << std::endl;*

*std::cout << maxValue(4.5, 7.2) << std::endl;*

*return 0;*

*}*

**2. Функции от по-висок ред и указатели към функции**

**Какво са функции от по-висок ред?**

Функциите от по-висок ред са такива, които приемат като аргумент друга функция или връщат функция като резултат. Това позволява **по-голяма гъвкавост и преизползваемост на кода**.

**Указатели към функции**

Указателят към функция е променлива, която съдържа адреса на функция и може да бъде използвана за извикване на тази функция.

***Пример:***

*#include <iostream>*

*int add(int a, int b) {*

*return a + b;*

*}*

*int main() {*

*int (\*funcPtr)(int, int) = add;*

*std::cout << funcPtr(3, 4) << std::endl; // Извежда: 7*

*return 0;*

*}*

**Задача 2:**

Напишете функция applyOperation, която приема два int аргумента и указател към функция, извършваща някаква операция върху тях (например събиране).

***Примерно решение:***

*#include <iostream>*

*int add(int a, int b) {*

*return a + b;*

*}*

*int applyOperation(int x, int y, int (\*operation)(int, int)) {*

*return operation(x, y);*

*}*

*int main() {*

*std::cout << applyOperation(10, 20, add) << std::endl; // 30*

*return 0;*

*}*

**3. std::function и неговото приложение**

**Какво е std::function?**

std::function е обект от **стандартната библиотека**, който позволява съхраняване и извикване на функции с определен подпис (signature). Той е по-гъвкав от указателите към функции, защото поддържа и **lambda** функции и методи на класове.

***Пример:***

*#include <iostream>*

*#include <functional>*

*int multiply(int a, int b) {*

*return a \* b;*

*}*

*int main() {*

*std::function<int(int, int)> operation = multiply;*

*std::cout << operation(5, 3) << std::endl; // 15*

*return 0;*

*}*

**Задача 3:**

Преработете applyOperation от предишната задача, така че вместо указател към функция, тя да използва std::function.

***Примерно решение:***

*#include <iostream>*

*#include <functional>*

*int applyOperation(int x, int y, std::function<int(int, int)> operation) {*

*return operation(x, y);*

*}*

*int main() {*

*std::cout << applyOperation(10, 20, [](int a, int b) { return a + b; }) << std::endl; // 30*

*return 0;*

*}*

**4. Lambda функции**

**Какво представляват lambda функциите?**

Lambda функциите са **анонимни функции**, които могат да бъдат дефинирани на място, предавани като аргументи и използвани без да се дефинират отделно.

***Пример:***

*#include <iostream>*

*int main() {*

*auto sum = [](int a, int b) { return a + b; };*

*std::cout << sum(3, 4) << std::endl; // 7*

*return 0;*

*}*

***Задача 4:***

*Използвайте lambda функция, за да подадете операция за умножение към applyOperation.*

***Примерно решение:***

*#include <iostream>*

*#include <functional>*

*int applyOperation(int x, int y, std::function<int(int, int)> operation) {*

*return operation(x, y);*

*}*

*int main() {*

*std::cout << applyOperation(10, 20, [](int a, int b) { return a \* b; }) << std::endl; // 200*

*return 0;*

*}*

**Седмица 4**

**Теоретична част: Class и модификатори за достъп**

**1. Какво е клас?**

Класът е основен градивен елемент на обектно-ориентираното програмиране (ООП). Той представлява шаблон, по който се създават обекти. Класовете съдържат:

* **Полета (атрибути)** – данни, свързани с обекта.
* **Методи** – функции, които дефинират поведението на обекта.
* **Конструктори** – специални методи, които се извикват при създаване на нов обект.

**2. Модификатори за достъп**

Модификаторите за достъп определят видимостта и достъпа до членовете на класа:

* public – достъпни от всяка точка в програмата.
* private – достъпни само в рамките на същия клас.
* protected – достъпни в рамките на класа и неговите наследници.

**3. Инкапсулация**

Инкапсулацията е концепция, която ограничава достъпа до данните на обекта и излага само необходимите методи. Тя осигурява:

* Скриване на детайлите на реализацията.
* Контрол върху достъпа до данните чрез методи (getter-и и setter-и).

**4. Черни кутии в ООП**

В обектно-ориентираното програмиране често използваме концепцията за "черна кутия" – обектите предоставят само определени интерфейси за взаимодействие, без да излагат вътрешната си реализация. Това позволява по-добра капсулация и модулност в кода.