

Języki i paradygmaty programowania

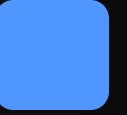
cz. II

Odc. 4

W dzisiejszym odcinku

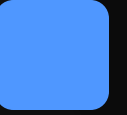


Przeładowania, Inline, C-ctor



- Wstęp do operatorów powtórka z C
- Przeładowania funkcji i operatorów
 - ()
 - []
 - ->
 - =
- Funkcje inline
- konstruktor kopiujący

Potwórka



- **Arytmetyczne**

- + - * / %

- **Porównania**

- == != > < >= <=

- **Logiczne**

- && || !

- **Przypisania**

- = += -= *= /= %=

- **Inkrementacji/Dekrementacji**

- ++ --

- **Bitowe**

- & | ^ ~ << >>

- **Adresu i Deferencji**


- & *

- **Warunkowe**

- ?: (Elvis)

- **Kropki i Strzałki**

- . ->



```
int a = 1;  
int b = 5;
```

```
int c = a+b;
```

```
int a[2] = {1, 1};  
int b[2] = {5, 5};
```

```
int c = a+b;
```

```
int a[2][2] = {{1,1},{1,1}};  
int b[2][2] = {{5,5},{5,5}};
```

```
int c[2][2] = a+b;
```

Overload przeładowanie/przeciążenie



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

```
struct Complex {
    Complex( double r, double i ) : re(r), im(i) {}
    Complex operator+( Complex &other );
    void Display( ) { cout << re << ", " << im << endl; }
private:
    double re, im;
};
```

```
// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
    return Complex( re + other.re, im + other.im );
}
```

```
int main() {
    Complex a = Complex( 1.2, 3.4 );
    Complex b = Complex( 5.6, 7.8 );
    Complex c = Complex( 0.0, 0.0 );

    c = a + b;
    c.Display();
}
```

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

```
struct Complex {
    Complex(double r, double i) : re(r), im(i) {}
    double re, im; // Oba pola są publiczne dla uproszczenia przykładu
};
```

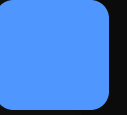
```
// Przeciążenie operatora + jako funkcji globalnej
Complex operator+(const Complex& lhs, const Complex& rhs) {
    return Complex(lhs.re + rhs.re, lhs.im + rhs.im);
}
```

```
int main() {
    Complex a = Complex( 1.2, 3.4 );
    Complex b = Complex( 5.6, 7.8 );
    Complex c = Complex( 0.0, 0.0 );

    c = a + b;
    c.Display();
}
```

Co wolno wojewodzie...
A nie, wojewodzie też nie.

- ?:
- ::
- .
- *
- sizeof
- typeid
- -> - POWIEDZMY...





You

make a list of non-overrideable operators in C++



ChatGPT

In C++, operators can typically be overridden for user-defined types (classes or structs). However, there are certain operators that cannot be overridden or overloaded. Here's a list of non-overrideable operators in C++:

1. **Scope resolution operator (::)**: Used to define the scope of a function or variable in C++. It cannot be overloaded.
2. **Member selection operator (.)**: Used to access members of a class or struct by using an object or a reference. It cannot be overloaded.
3. **Member selection pointer operator (->)**: Used to access members of a class or struct through a pointer. It cannot be overloaded.
4. **Sizeof operator (sizeof)**: Used to determine the size of a variable or datatype. It cannot be overloaded.
5. **Conditional operator (?:)**: Also known as the ternary operator, it is used for conditional expressions. It cannot be overloaded.
6. **Scope resolution operator for namespaces (::)**: Used to qualify names with the namespace in which they are defined. It cannot be overloaded.
7. **Dot-star operator (.*)**: Used for pointer-to-member access. It cannot be overloaded.
8. **Colon-colon operator (::)**: Used for nested-name-specifier in qualified lookup contexts. It cannot be overloaded.

These operators have specific functionalities and are not designed to be overloaded. They serve critical roles in the language's syntax and semantics.



You

give ma example of overriding -> operator



ChatGPT

In C++, the `->` operator is typically used to access members of an object through a pointer to that object. It is commonly seen in cases where you have a pointer to an object of a class or struct and you want to access its members. While you cannot directly overload the `->` operator for built-in pointer types, you can overload it for user-defined types, such as classes or structs.

Here's a simple example demonstrating how to override the `->` operator:

cpp

Copy code

```
#include <iostream>

class MySmartPointer {
private:
    int *ptr;

public:
    // Constructor
    MySmartPointer(int *p) : ptr(p) {}

    // Destructor
    ~MySmartPointer() {
        delete ptr;
    }

    // Overloading -> operator
    int* operator->() {
        return ptr;
    }
}
```



Message ChatGPT...



cudowne XD

w jednej linijce pisze że się nie da

ale jak trzeba przykład to spoko XD

xD

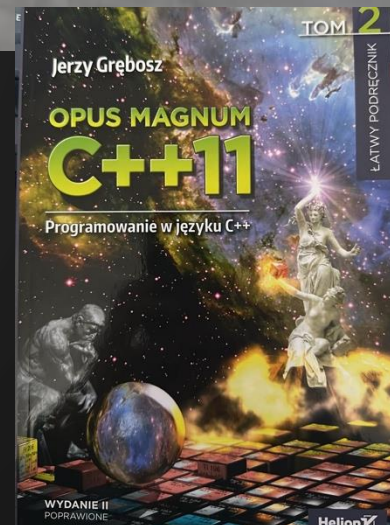
Czekaj

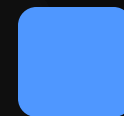
Wezmę Jurka

Zaczyna się dobrze...

27.7 Operator ->

Jeśli nie zrozumiesz tego paragrafu od razu, nie przejmuj się. Przeładowywanie tego operatora to już naprawdę hiszpańska szkoła jazdy. Nie żeby to było takie strasznie trudne, ale po prostu rzadko się to robi. Jednak mimo wszystko proszę o przeczytanie tego paragrafu, bo pomoże Ci on zrozumieć działanie bardzo przydatnej bibliotecznej klasy o nazwie `unique_ptr`.





"Narada na szczycie"

xD



Nie uczymy się. Bajlando

Overload ()



```
#include <iostream>

class MyFunction {
public:
    void operator()(int x) const {
        std::cout << "Function called with argument: " << x << std::endl;
    }
};

int main() {
    MyFunction func;
    func(5); // Calls operator() with argument 5
    return 0;
}
```

Overload []



```
#include <iostream>

class MyArray {
private:
    int arr[5] = {1, 2, 3, 4, 5};
public:
    int operator[](int index) const {
        return arr[index];
    }
};

int main() {
    MyArray arr;
    std::cout << "Element at index 2: " << arr[2] << std::endl; // Outputs: Element at index 2: 3
    return 0;
}
```

Overload ->



```
#include <iostream>

class MyClass {
public:
    void display() const {
        std::cout << "Hello from MyClass" << std::endl;
    }
};

class MyPointer {
private:
    MyClass* ptr;
public:
    MyPointer(MyClass* p) : ptr(p) {}
    MyClass* operator->() const {
        return ptr;
    }
};

int main() {
    MyClass obj;
    MyPointer ptr(&obj);
    ptr->display(); // Calls display() through overloaded operator->
    return 0;
}
```

Overload =



```
CopyDisable& operator = (const CopyDisable&) = delete;
```

```
IntArray& operator=(const IntArray& other) {  
    if (this != &other) { // Avoid self-assignment  
        // Deallocate current memory  
        delete[] array;  
  
        // Allocate new memory and perform deep copy  
        size = other.size;  
        array = new int[size];  
        memcpy(array, other.array, size * sizeof(int));  
  
        std::cout << "Assignment operator called" << std::endl;  
    }  
    return *this;  
}
```


Przeładowania funkcji



```
#include <iostream>

// Przeładowana funkcja add, dodająca dwie liczby całkowite
int add(int a, int b) {
    std::cout << "Using int version of add function" << std::endl;
    return a + b;
}

// Przeładowana funkcja add, dodająca dwa zmiennoprzecinkowe liczby
double add(double a, double b) {
    std::cout << "Using double version of add function" << std::endl;
    return a + b;
}

int main() {
    int sum_int = add(5, 3); // Wywołanie funkcji add z argumentami typu int
    std::cout << "Sum of integers: " << sum_int << std::endl;

    double sum_double = add(3.5, 2.5); // Wywołanie funkcji add z argumentami typu double
    std::cout << "Sum of doubles: " << sum_double << std::endl;

    return 0;
}
```

Przeładowania funkcji w klasie



```
#include <iostream>

class Printer {
public:
    // Przeładowana funkcja print dla typu int
    void print(int value) {
        std::cout << "Printing integer: " << value << std::endl;
    }

    // Przeładowana funkcja print dla typu double
    void print(double value) {
        std::cout << "Printing double: " << value << std::endl;
    }

    // Przeładowana funkcja print dla typu string
    void print(const std::string& value) {
        std::cout << "Printing string: " << value << std::endl;
    }
};

int main() {
    Printer printer;

    printer.print(10);           // Wywołanie funkcji print dla int
    printer.print(3.14);        // Wywołanie funkcji print dla double
    printer.print("Hello");      // Wywołanie funkcji print dla string

    return 0;
}
```

Funkcje inline



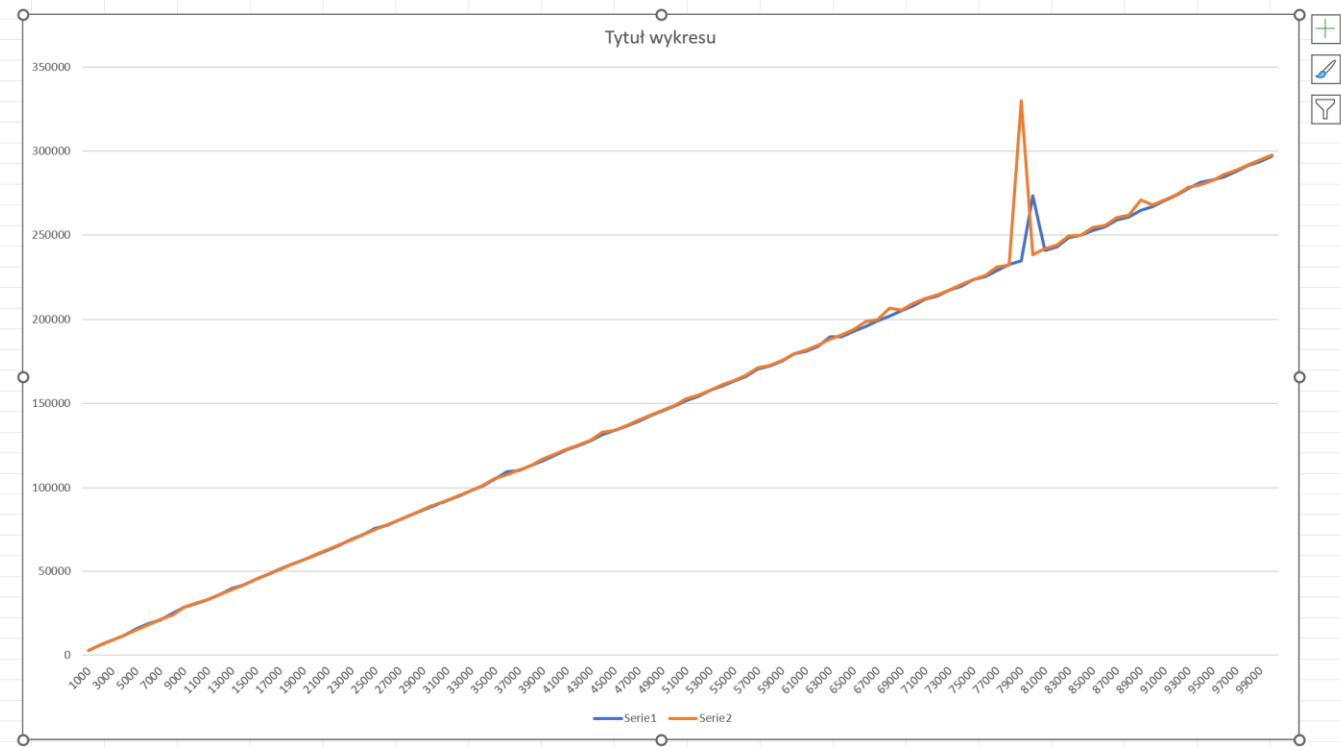
```
#include <iostream>

// Definicja funkcji inline
inline int dodaj(int a, int b) {
    return a + b;
}

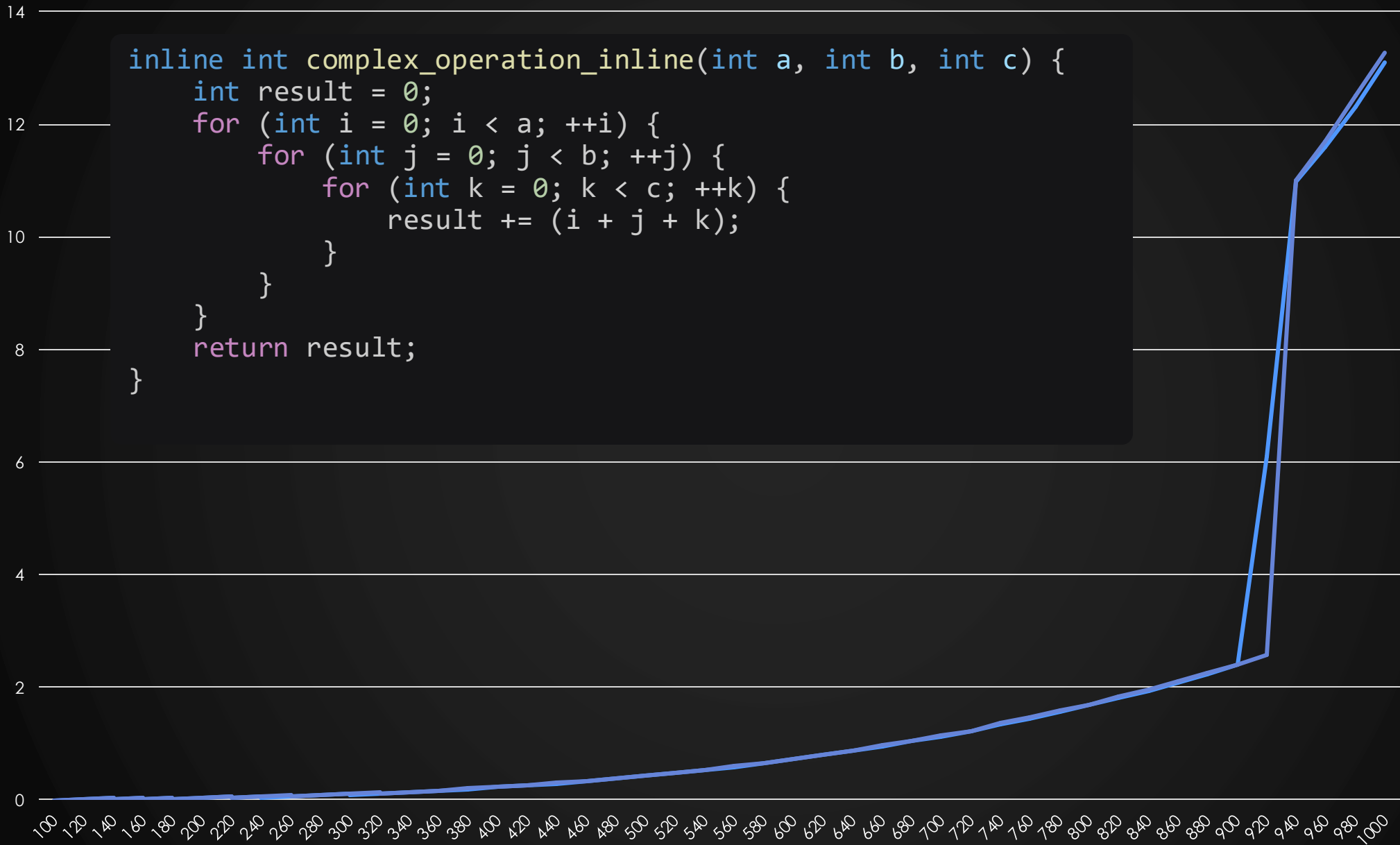
int main() {
    int x = 5, y = 10;
    int wynik = dodaj(x, y); // Wywołanie funkcji inline
    std::cout << "Wynik dodawania: " << wynik << std::endl;
    return 0;
}
```



1	1000	2986	3004
2	2000	6022	6107
3	3000	9108	9006
4	4000	12037	11891
5	5000	15729	15195
6	6000	18840	18081
7	7000	20931	21415
8	8000	25050	23878
9	9000	28340	28347
10	10000	30578	30925
11	11000	33077	33267
12	12000	36002	36047
13	13000	39541	38994
14	14000	41846	42001
15	15000	45015	45163
16	16000	47832	48082
17	17000	51140	51050
18	18000	53961	53978
19	19000	56710	56818
20	20000	59462	59737
21	21000	62518	62739
22	22000	65490	65685
23	23000	68770	68710
24	24000	71725	72005
25	25000	75390	74652
26	26000	77322	77793
27	27000	80397	80559
28	28000	83540	83538
29	29000	86320	86542
30	30000	89183	90005
31	31000	92373	92553
32	32000	95067	95430
33	33000	98059	98349
34	34000	100968	101364
35	35000	104954	105318
36	36000	109273	107665
37	37000	110077	110432
38	38000	112864	113148
39	39000	115887	116908
40	40000	119342	119909
41	41000	122470	122626
42	42000	124766	125211
43	43000	127744	128365
44	44000	131461	132987
45	45000	133826	134050
46	46000	136544	136997



```
inline int complex_operation_inline(int a, int b, int c) {  
    int result = 0;  
    for (int i = 0; i < a; ++i) {  
        for (int j = 0; j < b; ++j) {  
            for (int k = 0; k < c; ++k) {  
                result += (i + j + k);  
            }  
        }  
    }  
    return result;  
}
```



Konstruktor kopiujący



```
#include <iostream>

class Liczba {
private:
    int wartosc;

public:
    // Konstruktor
    Liczba(int val) : wartosc(val) {}

    // Konstruktor kopiujący
    Liczba(const Liczba& other) : wartosc(other.wartosc) {}

    // Metoda do wyświetlania wartości
    void wyswietl() {
        std::cout << "Wartość: " << wartosc << std::endl;
    }
};

int main() {
    // Tworzenie obiektu klasy Liczba
    Liczba liczba1(5);

    // Wywołanie konstruktora kopiującego
    Liczba liczba2 = liczba1;

    // Wyświetlenie wartości obiektu 1
    liczba1.wyswietl();

    // Wyświetlenie wartości obiektu 2
    liczba2.wyswietl();

    return 0;
}
```

Konstruktor kopiujący



```
#include <iostream>

class Punkt {
private:
    int x;
    int y;
public:
    // Konstruktor
    Punkt(int x_val, int y_val) : x(x_val), y(y_val) {}

    // Konstruktor kopiujący
    Punkt(const Punkt& other) : x(other.x), y(other.y) {}

    // Metoda do wyświetlania współrzędnych
    void wyswietl() {
        std::cout << "Współrzędne punktu: (" << x << ", " << y << ")" << std::endl;
    }
};

int main() {
    // Tworzenie obiektu klasy Punkt
    Punkt punkt1(3, 5);

    // Wywołanie konstruktora kopiującego
    Punkt punkt2 = punkt1;

    // Wyświetlenie współrzędnych obiektu 1
    punkt1.wyswietl();

    // Wyświetlenie współrzędnych obiektu 2
    punkt2.wyswietl();

    return 0;
}
```

Konstruktor kopiujący



```
#include <iostream>

class Liczba {
private:
    int wartosc;

public:
    // Konstruktor
    Liczba(int val) : wartosc(val) {}

    // Konstruktor kopiujący
    Liczba(const Liczba& other) : wartosc(other.wartosc) {}

    // Operator przypisania
    Liczba& operator=(const Liczba& other) {
        std::cout << "Wywołano operator przypisania" << std::endl;
        wartosc = other.wartosc;
        return *this;
    }

    // Metoda do wyświetlania wartości
    void wyswietl() {
        std::cout << "Wartość: " << wartosc << std::endl;
    }
};

int main() {
    // Tworzenie obiektu klasy Liczba
    Liczba liczba1(5);

    // Tworzenie obiektu klasy Liczba i przypisanie
    Liczba liczba2 = liczba1;

    // Wyświetlenie wartości obiektu 1
    std::cout << "Wartość obiektu 1: ";
    liczba1.wyswietl();

    // Wyświetlenie wartości obiektu 2
    std::cout << "Wartość obiektu 2: ";
    liczba2.wyswietl();

    return 0;
}
```


Konstruktor kopiujący



```
int main() {  
    // Tworzenie obiektu klasy Liczba  
    Liczba liczba1(5);  
  
    // Wywołanie konstruktora kopiującego  
    Liczba liczba2(liczba1);  
  
    // Wyświetlenie wartości obiektu 1  
    std::cout << "Wartość obiektu 1: ";  
    liczba1.wyswietl();  
  
    // Wyświetlenie wartości obiektu 2  
    std::cout << "Wartość obiektu 2: ";  
    liczba2.wyswietl();  
  
    return 0;  
}
```

Zadanko



Napisz program do zarządzania zamówieniami klientów księgarni.

Wymagania:

1. Stwórz klasę **Order**, która będzie reprezentować zamówienie klienta. Każde zamówienie będzie zawierać listę książek, które klient chce zakupić, wraz z odpowiadającymi ilościami.
2. Zdefiniuj operator **+=**, który pozwoli na dodanie kolejnej książki do zamówienia
3. zaimplementuj funkcję **calculate_total_price()**, która będzie obliczać całkowity koszt zamówienia, biorąc pod uwagę ceny każdej książki oraz ilości zamówionych egzemplarzy.
4. W funkcji **calculate_total_price()** uwzględnij opcję kosztów dostawy aktywowaną podaniem dodatkowego parametru