# Programowanie obiektowe

Polimorfizm funkcyjny

# Informacje wstępne

- **Przeciążanie operatorów** – specyficzny przypadek polimorfizmu, gdzie wybrane operatory traktowane są jako funkcje lub metody polimorficzne tzw. mają inne znaczenie w zależności od typu argumentów.

- Po co przeciążać operatory:
  - By wygodniej używać klas.
  - By ujednolicić operacje/dostęp do obiektów własnych typów danych.
  - By zmienić zachowanie istniejącego operatora (*używać tylko w razie absolutnej konieczności*).

Czy ktoś może podać przykład przeciążonego operatora?

# Przykład zastosowania operatorów

Załóżmy, że istnieją następujące obiekty klasy Liczba:

```
Liczba l1(20);
Liczba l2(5);
Liczba l3(4);
```

Dodatkowo, niech możliwa jest metoda **dodaj()**, która sumuje liczby ze sobą oraz metoda **pomnoz()**, która generuje ich iloczyn. Dodawanie i mnożenie 3 liczb można zapisać tak:

```
int suma = dodaj(l1, pomnoz(l2, l3));
```

Czy nie lepiej zapisać to jako:

```
int suma = l1 + l2 * l3;
```

# Przeciążanie operatorów

- Istnieją 2 warianty przeciążania operatorów:
  - Operator zdefiniowany jako składowa klasy.
  - Operator nie będący składową klasy.

- Ogólny schemat przeciążania operatorów:

```
typ_zwracany operator@(lista_parametrów)
{
    // ... definicja
}
```

- Zatem wywołanie dla obiektów klasy Liczba może wyglądać tak:
  - `int suma = l1 + l2;`

Lub:

  - `int suma = l1.operator+(l2);`

# Liczba.h

```
1   #ifndef LICZBA_H
2   #define LICZBA_H
3
4   class Liczba
5   {
6       public:
7           Liczba();
8           Liczba(int w_poczatkowa);
9
10          int Getwartosc() { return m_wartosc; }
11          void Setwartosc(int val) { m_wartosc = val; }
12          int operator+(Liczba l2);
13
14      protected:
15
16      private:
17          int m_wartosc;
18  };
19
```

# Liczba.cpp

```
1   #include "Liczba.h"
2
3   Liczba::Liczba()
4   {
5       m_wartosc = 0;
6   }
7
8   Liczba::Liczba(int w_poczatkowa)
9   {
10      m_wartosc = w_poczatkowa;
11  }
12
13
14  int Liczba::operator+(Liczba l2)
15  {
16      return m_wartosc + l2.Getwartosc();
17  }
```

# Main.cpp

- Przeciążony operator jako metoda składowa klasy:

```
1   #include <iostream>
2   #include "Liczba.h"
3
4   using namespace std;
5
6   int main()
7   {
8       Liczba l1(5);
9       Liczba l2(10);
10      int suma = l1 + l2;
11      cout << "Suma to " << suma << endl;
12      return 0;
13  }
```

# Przeciążanie operatorów – problemy

```
1   #include <iostream>
2   #include "Liczba.h"
3
4   using namespace std;
5
6
7   int main()
8   {
9       Liczba l1(5);
10      Liczba l2(10);
11
12      //Można tak
13      int suma = l1 + 5;
14      //Ale tak już nie
15      int suma2 = 5 + l1;
16
17      cout << "Suma to " << suma << endl;
18      return 0;
19  }
```

# Przeciążanie operatorów – postać funkcyjna

- Pozbywamy się deklaracji operatora jako składowej klasy i definiujemy go w postaci funkcyjnej.

```
1   #include <iostream>
2   #include "Liczba.h"
3
4   using namespace std;
5
6   int operator+(Liczba l1, Liczba l2)
7   {
8       return l1.Getwartosc() + l2.Getwartosc();
9   }
10
11  int main()
12  {
13      Liczba l1(5);
14      Liczba l2(10);
15
16      //Można tak
17      int suma = l1 + 5;
18      //Teraz też tak
19      int suma2 = 5 + l1;
20
21      cout << "Suma to " << suma << " a Suma2 to " << suma2 << endl;
22      return 0;
23  }
```

# Czy to jest poprawne?

```
1   #include <iostream>
2   #include "Liczba.h"
3
4   using namespace std;
5
6   int operator+(Liczba l1, Liczba l2)
7   {
8       return l1.Getwartosc() + l2.Getwartosc();
9   }
10
11  int main()
12  {
13      Liczba l1(5);
14      Liczba l2(10);
15      Liczba l3(7);
16
17      int suma = l1 + l2 + l3;
18      cout << "Suma to " << suma;
19      return 0;
20  }
```

```
l1.operator+(l2.operator+(l3));
```

# Limity i ograniczenia przy przeciążaniu operatorów

1. Przeciążony operator musi mieć jeden operand typu zdefiniowanego przez użytkownika. Zapobiega to przeciążaniu operatorów dla typów standardowych.
2. Nie można zmieniać składni operatora ani jego pierwszeństwa – przykładowo operator modulo % musi zostać dwuargumentowy.
3. Nie można tworzyć własnych symboli operatorów.
4. Przeciążony operator nie może mieć parametrów domyślnych.
5. Nie wszystkich operatorów można przeciążać.

# Operatory, których nie można przeciążać.

- `?:` (operator warunkowy)
- `.` (operator dostępu do składowej)
- `.*` (wskaźnik na składową)
- `::` (operator zasięgu)
- `sizeof` (informacja o rozmiarze)
- `typeid` (informacja o typie)
- `static_cast` (operator rzutowania)
- `const_cast` (operator rzutowania)
- `reinterpret_cast` (operator rzutowania)
- `dynamic_cast` (operator rzutowania)

# Przeciążanie operatora wstawiania

```
1   #include <iostream>
2   #include "Liczba.h"
3
4   using namespace std;
5
6
7   void operator<<(ostream & os, Liczba l1)
8   {
9       os << l1.Getwartosc();
10  }
11
12  int main()
13  {
14      Liczba l1(5);
15
16      cout << l1;
17      return 0;
18  }
```

# Przeciążanie operatora wstawiania – problemy

```
1   #include <iostream>
2   #include "Liczba.h"
3
4   using namespace std;
5
6
7   void operator<<(ostream & os, Liczba l1)
8   {
9       os << l1.Getwartosc();
10  }
11
12  int main()
13  {
14      Liczba l1(5);
15      Liczba l2(10);
16
17      cout << l1 << l2;
18      return 0;
19  }
```

# Przeciążanie operatora wstawiania – rozwiązanie

```
1   #include <iostream>
2   #include "Liczba.h"
3
4   using namespace std;
5
6
7   ostream& operator<<(ostream & os, Liczba l1)
8   {
9       os << l1.Getwartosc();
10      return os;
11  }
12
13  int main()
14  {
15      Liczba l1(5);
16      Liczba l2(10);
17
18      cout << l1 << " oraz " << l2;
19      return 0;
20  }
```

# Trochę optymalizacji

```
1   #include <iostream>
2   #include "Liczba.h"
3
4   using namespace std;
5
6
7   ostream& operator<<(ostream & os, const Liczba & l1)
8   {
9       os << l1.Getwartosc();
10      return os;
11  }
12
13  int main()
14  {
15      Liczba l1(5);
16      Liczba l2(10);
17
18      cout << l1 << " oraz " << l2;
19      return 0;
20  }
21
```

Ale pamiętaj o dodaniu **const** do Getwartosc() w Liczba.h

# Ćwiczenia

1. Na podstawie uprzednio stworzonej klasy **DataZKontrola**, wykorzystując mechanizm przeciążenia operatorów, dodaj do niej możliwość wyświetlenia na ekran aktualnego stanu obiektu oraz sumowanie dwóch dat (tylko biorąc pod uwagę rok).
2. Bazując na klasie **Punkt3D**, dodaj podstawowe operacje matematyczne dla dwóch punktów takie jak ich:
   - Dodawanie.
   - Odejmowanie.
   - Mnożenie.
   - Dzielenie.
   - Dodatkowo zaimplementuj możliwość mnożenia wszystkich współrzędnych punktu przez dowolną liczbę całkowitą.

# Ćwiczenia

3. Wykorzystując klasę **Lista**, dokonaj przeciążenia operatora porównania tak, by można było sprawdzić, czy dwie listy zawierają dokładnie te same elementy, ale ustawione w dowolnej kolejności.
4. Na podstawie klasy **Figura**, dokonaj przeciążenia operatorów dodawania i odejmowania, by umożliwiały wyznaczenie sumy bądź różnicy pół powierzchni dowolnych figur geometrycznych (dziedziczących z klasy Figura).