



# KURS JĘZYKA C++

## 4. PRZECIĄŻANIE OPERATORÓW



# SPIS TREŚCI

- Funkcje zaprzyjaźnione
- Przeciążanie operatorów
- Operatory składowe w klasie
- Zaprzyjaźnione funkcje operatorowe
- Operatory predefiniowane
- Niestatyczne operatory składowe
- Operatory `new` i `delete`
- Operatory strumieniowe `<< i >>`

# FUNKCJE ZAPRZYJAŻNIONE

- Problem z kwiatkami w domu w czasie dalekiej podróży służbowej.
- Funkcja, która jest przyjacielem klasy, ma dostęp do wszystkich jej prywatnych i chronionych składowych.
- To klasa deklaruje, które funkcje są jej przyjaciółmi.
- Deklaracja przyjaźni może się pojawić w dowolnej sekcji i jest poprzedzona słowem kluczowym `friend`.

# FUNKCJE ZAPRZYJAŻNIONE

## ■ Przykład klasy z funkcją zaprzyjaźnioną:

```
// klasa z funkcją zaprzyjaźnioną
class pionek
{
    int x, y;
    // ...
    friend void raport(const pionek &p);
};
// funkcja, która jest przyjacielem klasy
void raport (const pionek &p)
{
    cout << "(" << p.x << ", " << p.y << ")";
}
```

# FUNKCJE ZAPRZYJAŻNIONE

- Nie ma znaczenia, w której sekcji (prywatnej, chronionej czy publicznej) pojawi się deklaracja przyjaźni.
- Funkcja zaprzyjaźniona z klasą nie jest jej składową, nie może używać wskaźnika `this` w stosunku do obiektów tej klasy.
- Jedna funkcja może się przyjaźnić z kilkoma klasami.
- Istotą przyjaźni jest dostęp do niepublicznych składowych w klasie – sensowne jest deklarowanie przyjaźni, gdy dana funkcja pracuje z obiektami tej klasy.

# FUNKCJE ZAPRZYJAŻNIONE

- Można także umieścić w klasie nie tylko deklarację funkcji zaprzyjaźnionej, ale również jej definicję; tak zdefiniowana funkcja:
  - jest nadal tylko przyjacielem klasy;
  - jest *inline*;
  - może korzystać z typów zdefiniowanych w klasie.
- Funkcją zaprzyjaźnioną może być funkcja składowa z innej klasy.

# KLASY ZAPRZYJAŻNIONE

- Możemy w klasie zadeklarować przyjaźń z inną klasą, co oznacza, że wszystkie metody tej innej klasy są zaprzyjaźnione z klasą pierwotną.

- Przykład:

```
class A
{
    friend class B;
    // ...
};
```

- Przyjaźń jest jednostronna.
- Przyjaźń nie jest przechodnia.
- Przyjaźni się nie dziedziczy.

# KLASY ZAPRZYJAŹNIONE

- Dwie klasy mogą się przyjaźnić z wzajemnością:

```
class A;  
class B;  
  
class B {  
    friend class A;  
    // ...  
};  
class A {  
    friend class B;  
    // ...  
};
```



# PO CO PRZECIĄŻAĆ OPERATORY?

- Porównaj dwa wyrażenia:

$$y = a * x + b ;$$

$$y = \text{dodaj}(\text{pomnoz}(a, x), b) ;$$

- A teraz wyobraź sobie funkcyjny zapis takiego wyrażenia:

$$y = (a * c - b * d) / (a * a + b * b) ;$$

- Operatory tylko upraszczają notację wyrażeń.

# OGÓLNE ZASADY PRZECIĄŻANIA OPERATORÓW

- Można tylko przeciążać operatory, nie wolno definiować nowych.
- Przy przeciążaniu operatora nie można zmienić jego priorytetu, arności ani łączności.
- Co najmniej jeden z argumentów przeciążanego operatora musi się odnosić do klasy (nie wolno zmieniać znaczenia operatorów w stosunku do typów podstawowych).
- Nie wolno używać argumentów domyślnych w operatorach.

# PRZECIAŻANIE OPERATORÓW

■ Nazwa funkcji operatorowej to `operator @`, gdzie `@` to symbol (nazwa) operatora.

■ Można deklarować funkcje definiujące znaczenie następujących operatorów:

`+ - * / % ^ & | << >>`  
`+= -= *= /= %= ^= &= |= <<= >>=`  
`= ~ ! < > <= >= == != ,`  
`&& || ++ -- -> ->* [] ()`  
`new new[] delete delete[]`

■ Można definiować zarówno operatory dwuargumentowe jak i jednoargumentowe (prefiksowe i postfiksowe).

# PRZYKŁAD PRZECIĄŻENIA OPERATORA

- Przykład klasy pamiętającej liczbę zespoloną, dla której przeciążymy operator dodawania:

```
class comp
{
public:
    const double re, im;
public:
    comp (double r=0, double i=0) : re(r), im(i) {}
    comp (const comp &c) : re(c.re), im(c.im) {}
};
```

- Przykład operatora dodawania dla obiektów z liczbami zespolonymi:

```
comp operator + (comp a, comp b)
{
    return comp(a.re+b.re, a.im+b.im);
}
```

- Przykład użycia operatora dodawania liczb zespolonych:

```
comp a(2), b(3,5), c = a + b;
```

# PRZECIĄŻANIE OPERATORÓW

- Nie można definiować następujących operatorów:
  - ? : (operator warunkowy)
  - :: (rezolucja zasięgu)
  - . (wybór składowej)
  - . \* (wybór składowej za pomocą wskaźnika do składowej)
- Nie można też przeciążyć operatora, który podaje rozmiar obiektu `sizeof` oraz operatora rozmieszczenia danych w pamięci `alignof`.
- Nie wolno przeciążać operatorów rzutowania: `static_cast`, `dynamic_cast`, `const_cast` i `reinterpret_cast`.
- Nie wolno definiować operatorów `#` i `##`, które są poleceniami dla prekompilatora.

# ZAPRZYJAŻNIONE FUNKCJE OPERATOROWE

- Bardzo często funkcje operatorowe sięgają do ukrytych składowych w klasie – wtedy wygodnie jest zadeklarować w klasie przyjaźń z takim operatorem.

- Przykład:

```
class comp {
    friend comp operator + (comp a, comp b);
    double re, im;
public:
    comp (double r=0, double i=0) : re(r), im(i) {}
    // ...
};
comp operator + (comp a, comp b) {
    return comp(a.re + b.re, a.im + b.im);
}
...
comp x(3, 7), y(5);
x = x + y;
x = x + 8.5;
x = -7.5 + x;
```

# OPERATORY SKŁADOWE W KLASIE

- Można zdefiniować operator jako funkcję składową w klasie – wtedy pierwszym niejawnym argumentem będzie obiekt tej klasy.

- Przykład:

```
class comp {
    double re, im;
public:
    comp (double r=0, double i=0) : re(r), im(i) {}
    // ...
    comp operator- (comp b);
    comp operator- ();
};
comp comp::operator- (comp b) {
    return comp(re - b.re, im - b.im);
}
comp comp::operator- () {
    return comp(-re, -im);
}
...
comp x(3, 7), y(5);
x = -x - y;
x = x - 8.5;
// x = 7.5 - x; // błąd
```

# SYMBOLICZNE I FUNKCYJNE WYWOŁANIE FUNKCJI OPERATOROWEJ

■ Niech dana będzie funkcja operatorowa `operator@`.

Wtedy możemy ją wywołać na dwa sposoby:

`x @ y` // wywołanie symboliczne

`operator@(x, y)` // wywołanie funkcyjne

■ Niech dana będzie składowa funkcja operatorowa

`operator@`. Wtedy możemy ją wywołać na dwa sposoby:

`x @ y` // wywołanie symboliczne

`x.operator@(y)` // wywołanie funkcyjne



# OPERATORY PREDEFINIOWANE

- Jest kilka operatorów, których znaczenie jest tak intuicyjne, że są one automatycznie wygenerowane dla każdej klasy:
  - przypisanie `=`,
  - jednoargumentowy operator pobrania adresu `&`,
  - sekwencja kolejnych wyrażeń `,` (przecinek),
  - tworzenie i usuwanie obiektów `new`, `new []`, `delete`, `delete []`.
- Można zdefiniować własne wersje wymienionych operatorów, jeśli chcemy zmienić ich domyślne zachowanie.

# NIESTATYCZNE OPERATORY SKŁADOWE

- Istnieją cztery operatory, które muszą być niestatycznymi operatorami składowymi:
  - przypisanie =,
  - indeksowanie [ ],
  - wywołanie funkcji ( ),
  - odwołanie do składowej ->.

# OPERATOR PRZYPISANIA =

- Jeśli nie zdefiniujemy przypisania kopiującego, to wygeneruje go kompilator (o ile nie ma w naszej klasie pól stałych).

- Postać operatora przypisania kopiującego:

```
K & K::operator= (const K &k) { /*...*/ }  
K & K::operator= (K &k) { /*...*/ }
```

- Domyślny operator przypisania kopiującego kopiuje składnik po składniku. Ale czasami takie kopiowanie nie jest dobre!

- Operator przypisania można przeciążać.

- Cechy prawidłowo napisanego operatora przypisania:

- nie zmienia stanu wzorca, z którego kopiuje;
- sprawdza, czy nie kopiuje sam na siebie;
- likwiduje bieżące zasoby (podobnie do destruktora);
- tworzy nowy stan obiektu na podobieństwo wzorca (podobnie jak konstruktor kopiujący).

- Przykład:

```
K & K::operator= (const K &k)  
{  
    if (&k==this) return *this;  
    this->~K();  
    // kopiowanie (głębokie) stanu z obiektu k  
    return *this;  
}
```

# OPERATOR INDEKSOWANIA [ ]

- Operator odwołania do tablicy można zaadoptować do odwoływania się do elementów kolekcji wewnątrz obiektu.
- Aby odwołanie indeksowe mogło stać po obu stronach operatora przypisania musimy zwracać referencję do elementu kolekcji.
- Indeksować można dowolnym typem (niekoniecznie `int`).

## ■ Przykład:

```
double comp::operator[] (bool b) const {  
    return b ? re : im;  
}  
double& comp::operator[] (bool b) {  
    return b ? re : im;  
}
```

# OPERATOR WYWOŁANIA FUNKCJI ()

- Operator wywołania funkcji () może mieć dowolną liczbę argumentów (również więcej niż dwa).
- Operator wywołania funkcji wywołujemy na obiekcie.
- Operator ten może mieć argumenty domniemane.
- Operator ten można przeciążać wiele razy w klasie.
- Wywołuje się go na rzecz jakiegoś obiektu. Przykład:

```
class K;  
K a;  
// ...  
a(); // a.operator() ();  
// ...  
a(1,2,3); // a.operator() (1,2,3);
```

# OPERATOR WSKAZYWANIA NA SKŁADOWĄ ->

- Operator ten wywołujemy na obiekcie (a nie na wskaźniku do danego obiektu).
- Operator ten musi zwracać albo wskaźnik albo obiekt takiej klasy, który ma przeładowany operator ->.
- Wywołanie:  
`obiekt->skladowa`  
Interpretacja wywołania:  
`(obiekt.operator->())->skladowa`

# POSTINKREMENTACJA I POSTDEKREMENTACJA

- Operatory ++ i -- mogą być zarówno prefiksowe jak i postfiksowe; prefiksowe operatory ++ i -- definiuje się jako jednoargumentowe (naturalna definicja) a postfiksowe jako dwuargumentowe:

```
class K
{
public:
    // operatory prefiksowe
    K & operator ++ ();
    K & operator -- ();
    // operatory postfiksowe
    K operator ++ (int);
    K operator -- (int);
    // ...
};
```

# OPERATORY NEW I NEW [ ] ORAZ DELETE I DELETE [ ]

- W klasie można zdefiniować własne operatory `new` i `delete`; jeśli są one zdefiniowane to kompilator użyje właśnie ich (a nie globalnych operatorów) do przydzielania i zwalniania pamięci.

- Definicja operatorów `new` i `delete` musi wyglądać następująco:

```
class K
{
public:
    // operator new
    static void* operator new (size_t s);
    static void* operator new[] (size_t s);
    // operator delete
    static void operator delete (void *p);
    static void operator delete[] (void *p);
    // ...
};
```

- W definicji własnych operatorów `new` i `delete` można odwoływać się do globalnych operatorów przydzielania i zwalniania pamięci `::new` i `::delete`.



# OPERATORY `new` I `new []`

- Operator `new` ma przydzielić pamięć dla pojedynczego obiektu a operator `new []` dla tablicy obiektów.
- Operatory `new` i `new []` muszą być statyczne w klasie.
- Operatory `new` i `new []` zwracają jako wynik wartość typu `void*`.
- Operatory `new` i `new []` przyjmują jako argument wartość typu `size_t` (w przypadku `new` ma to być rozmiar jednego obiektu a w przypadku `new []` rozmiar wszystkich obiektów łącznie); argument ten jest do tych operatorów przekazywany niejawnie (za pomocą operatora `sizeof`).
- Gdy zabraknie pamięci należy zgłosić wyjątek `bad_alloc` z pliku nagłówkowego `<new>`.

## OPERATORY `delete` I `delete []`

- Operator `delete` ma zwolnić pamięć dla pojedynczego obiektu a operator `delete []` dla tablicy obiektów.
- Operatory `delete` i `delete []` muszą być statyczne w klasie.
- Operatory `delete` i `delete []` nie zwracają wyniku (są typu `void`).
- Operatory `delete` i `delete []` przyjmują jako argument wskaźnik typu `void*`.
- W przypadku argumentu będącego wskaźnikiem pustym `nullptr`, nie należy podejmować żadnych akcji.

# GLOBALNE OPERATORY `new` I `new []` ORAZ `delete` I `delete []`

- Można zdefiniować własne wersje globalnych operatorów `new` i `new []` oraz `delete` i `delete []` ale:
  - w ten sposób całkowicie niszczymy oryginalne wersje tych operatorów;
  - operator `::new` jest używany w bibliotekach standardowych do tworzenia obiektów globalnych (takich jak `cin` czy `cout`) jeszcze przed uruchomieniem funkcji `main()`;
  - najczęściej własne definicje tych operatorów to błąd projektowy, który może doprowadzić do katastrofy w działaniu programu...

## OPERATORY `new []` | `delete []`

- Operator `new []` przydziela pamięć dla tablicy obiektów. Wszystkie obiekty w nowo utworzonej tablicy będą zainicjalizowane konstruktorem domyślnym (pamiętaj o zdefiniowaniu konstruktora domyślnego w klasie, której obiekty będą występować w tablicach).
- Operator `delete []` zwalnia pamięć przydzieloną dla tablicy obiektów. Przed zwolnieniem tej pamięci dla wszystkich obiektów zostanie wykonany destruktor.

# OPERATORY << | >>

## DO PRACY ZE STRUMIENIAMI

- Wygodnie jest zdefiniować operatory << i >> do pracy ze strumieniami; aby można było pracować z takimi operatorami w sposób kaskadowy powinny one być zdefiniowane jako funkcje zewnętrzne w stosunku do klasy i zwracać referencję do strumienia, na którym działają:

```
class K
{
    // operator czytający dane ze strumienia
    friend
    istream& operator >>
    (istream &is, K &k);
    // operator piszący dane do strumienia
    friend
    ostream& operator <<
    (ostream &os, const K &k);
    // ...
};
```