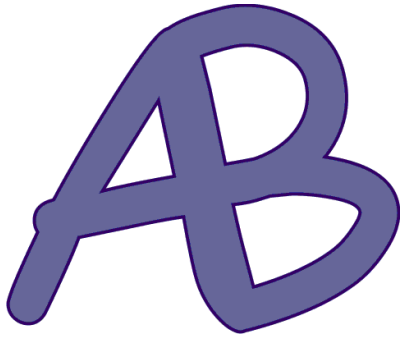


dr Artur Bartoszewski
Katedra Informatyki i teleinformatyki
Uniwersytet Radomski

Przeciążanie operatorów

Przeciążanie operatorów

Przeciążanie operatorów (ang. operator overloading), to definiowanie operatorów dla obiektów własnych klas.

Użytkownik może sam zdefiniować swój własny operator ale przynajmniej jeden z jego operandów, musi typem użytkownika (czyli klasą lub strukturą)

Kiedy w programie użyty jest operator (+, -, *, /,=,->,<<, itp.), jest wywoływana specjalna metoda, która zajmuje się wykonaniem żądanego działania – operator tego działania.

Metodę tę, można, jak każdą funkcję, przeciążyć (tzn. napisać jej nową wersję) pod warunkiem, że będzie się ona różniła listą parametrów od już istniejących wersji.

Dlatego właśnie nie można przeciążać operatorów pracujących tylko na typach wbudowanych – wszystkie możliwe kombinacje są już zapisane w standardzie języka. Pole do popisu dają dopiero operatory pracujące na obiektach klas.

Przeciążanie operatorów

Przeciążenia operatora dokonuje się definiując metodę operatorową, której nazwa składa się:

- ze słowa kluczowego operator
- po którym następuje symbol operatora, np.: + , - , * , itd.

```
typ_zwracany operator@ (argumenty)
//w miejsce znaku @ wstawiamy operator np.: +
{
// treść metody
}
```

- Przynajmniej jeden z argumentów, musi być obiektem klasy.
- Musi to być obiekt, a nie wskaźnik do obiektu.

Przeciążanie operatorów

Przeciążać możemy operatory:

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

Zaznaczone na zielono to operatory, które zawsze są przeciążone.
Jeżeli tworząc klasę nie zdefiniujemy ich sami zrobi to za nas kompilator.

Przeciążanie operatorów

Przeciążając operator sami decydujemy jak ma działać, jednak w pewnych granicach.

Nie można zmieniać:

- priorytetu operatorów,
- liczby parametrów operatorów – jeżeli operator posiada jeden czy dwa parametry, to tak musi zostać,
- łączności operatorów – czyli tego, czy operator łączy się z argumentem z lewej, czy z prawej strony

Przeciążanie operatorów

Przeciążać nie można operatorów:

. (operator dostępu do składowych)

.* (operator wskaźnika do składowej)

:: (operator zakresu)

sizeof

typeid

Operatory, które mogą być zdefiniowane wyłącznie jako metody wewnątrz klasy:

=	[ ]	->

Przeciążenie może nadać operatorowi dowolne znacznie, nie ma też ograniczeń co do wartości zwracanej przez operator (wyjątkami są operatory new i delete).

Przeciążanie operatorów

Funkcje operatorowe mogą być zdefiniowane jako:

1. Globalne funkcje programu
2. Funkcje zaprzyjaźnione z klasą
3. Metody klasy

Przeciążanie operatorów

1. Funkcje operatorowe zdefiniowane jako

globalne funkcje programu

(poza klasą)

Przeciążanie operatorów

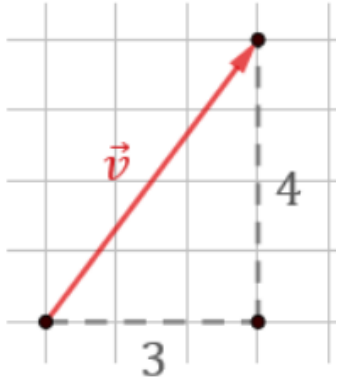
Funkcje operatorowe zdefiniowane jako globalne funkcje programu.

```
class klasa
{
};

klasa operator+(klasa a, klasa b)
{
    //treść metody
}
```

- Funkcja operatorowa zdefiniowana jako globalna, przyjmuje tyle argumentów na ilu pracuje operator.
- Przynajmniej jeden z argumentów musi być typu zdefiniowanego przez użytkownika (klasy) – nie ma znaczenia który.
- Parametry nie mogą być domyślne
- Funkcja musi mieć dostęp do pól klasy (bezpośrednio, lub za pomocą get-erów i set-erów)

Operatory zdefiniowane jako funkcje globalne



Dla zilustrowania omawianych zagadnień posłużymy się klasą „Wektor” przechowującą współrzędne wektora [x , y] wektory będziemy dodawać, odejmować, mnożyć itp..

```
class Wektor
{
public:
    double x, y;
    // Konstruktor
    Wektor(double x = 0.0, double y = 0.0) : x(x), y(y) {}

    // Metoda wypisująca współrzędne
    void wypisz() const
    {
        cout << "[" << x << ", " << y << "]\n";
    }
};
```

Operatory zdefiniowane jako funkcje globalne

Operatory dwuargumentowe

```
Wektor operator+(const Wektor &w1, const Wektor &w2)
{
    return Wektor(w1.x + w2.x, w1.y + w2.y);
}

Wektor operator-(const Wektor &w1, const Wektor &w2)
{
    return Wektor(w1.x - w2.x, w1.y - w2.y);
}
```

Przekazywanie parametrów przez referencję nie jest w tym wypadku niezbędne, lecz optymalizuje wykorzystanie pamięci - w trakcie wywołania funkcji nie powstają kopie obiektów.

Z kolei przekazywanie parametrów jako stałe „const” zabezpiecza przed ich przypadkową modyfikacją wewnątrz funkcji

```
int main()
{
    Wektor w1(1.0, 2.0);
    Wektor w2(3.0, 4.0);
    Wektor w3 = w1 + w2; // Przeciążony operator +
    Wektor w4 = w1 - w2; // Przeciążony operator -
    return 0;
}
```

Operatory zdefiniowane jako funkcje globalne

Przemienność operatorów

W przypadku, gdy po obu stronach operatora stoją parametry różnych typów.
Pojawia się problem z przemiennością działań (np. mnożenia wektora przez skalar).

Rozwiązaniem jest przygotowanie **dwóch wersji** funkcji operatorowej.

```
Wektor operator*(const Wektor &w, double scalar)
{
    return Wektor(w.x * scalar, w.y * scalar);
}
Wektor operator*(double scalar, const Wektor &w)
{
    return Wektor(w.x * scalar, w.y * scalar);
}
```

```
// Przeciążony operator *
Wektor wynik = w1 * 2.0;
Wektor wynik = 2.0 * w1;
```

Jeśli po lewej stronie operatora będzie stał typ wbudowany, to musi być funkcją globalną - nie może być metodą klasy.

Operatory zdefiniowane jako funkcje globalne

Skrócone operatory działań

W przypadku skróconych operatorów działań funkcja operatorowa nie zwraca wartości lecz musi mieć prawo modyfikacji argumentu stojącego po lewej stronie.

Stąd jako pierwszy argument przesyłamy referencję do obiektu oraz nie może on być stałą

```
void operator+=(Wektor &w1, const Wektor &w2)
{
    w1.x += w2.x;
    w1.y += w2.y;
}

void operator+=(Wektor &w1, double scalar)
{
    w1.x += scalar;
    w1.y += scalar;
}
```

```
// Przeciążony operator +=
w2+= w1;
w2+= 5.0;
```

W tym wypadku przemienność działań nie ma sensu – dopuścilibyśmy sytuacje: 2 += w1;

Operatory zdefiniowane jako funkcje globalne

Operatory strumieni wejścia-wyjścia

Aby ułatwić sobie pracę z obiektami zawierającymi wiele danych możemy przeciążyć operatory strumieni wejścia wyjścia. Dzięki temu będziemy mogli wypisywać na ekran i wczytywać dane z obiektów tak jak ze zmiennych typów wbudowanych..

```
ostream &operator<<(ostream &os, const Wektor &w)
{
    os << "[" << w.x << ", " << w.y << "]";
    return os;
}
istream &operator>>(istream &is, Wektor &w)
{
    is >> w.x >> w.y; // Strum wejściowe w formacie "x y" (separator to spacja lub enter)
    return is;
}
```

Wczytanie wartości sformatowanej w inny sposób jest oczywiście możliwe.
Ze strumienia wczytujemy łańcuch znaków i wyciągamy z niego wartości dla poszczególnych pół.

Przeciążanie operatorów

2. Funkcje operatorowe zaprzyjaźnione z klasą

Aby funkcja operatorowa zdefiniowana jako funkcja globalna miała dostęp do składników prywatnych klasy należy zadeklarować ją jako **zaprzyjaźnioną z klasą**.

Funkcje operatorowe zaprzyjaźnione z klasą

```
class Wektor
{
private:
    double x, y;
public:
    // Konstruktor
    Wektor(double x = 0.0, double y = 0.0) : x(x), y(y) {}
    // metody dostępowe
    double getX() const { return x; }
    double getY() const { return y; }
    void setX(double x) { this->x = x; }
    void setY(double y) { this->y = y; }
    // Metoda wypisująca współrzędne
    void wypisz() const
    {
        cout << "[" << x << ", " << y << "]\n";
    }
};
```

W poprzednich przykładach pola klasy były publiczne więc funkcje operator owe miały do nich swobodny dostęp.

Tym pola są prywatne

W poprzednich przykładach pola klasy były publiczne więc funkcje operator owe miały do nich swobodny dostęp.

Tym pola są prywatne

Funkcje operatorowe zaprzyjaźnione z klasą

```
class Wektor
{
private:
    double x, y;
public:
    // Konstruktor
    Wektor(double x = 0.0, double y = 0.0) : x(x), y(y) {}
    // metody dostępowe
    double getX() const { return x; }
    double getY() const { return y; }
    void setX(double x) { this->x = x; }
    void setY(double y) { this->y = y; }
};
```

Dostęp do pól prywatnych uzyskać możemy za pomocą metod dostępowych set i get. jednak metody te nie zawsze istnieją ….

```
Wektor operator+(const Wektor &w1, const Wektor &w2)
{
    return Wektor(w1.getX() + w2.getX(), w1.getY() + w2.getY());
}
```

Funkcje operatorowe zaprzyjaźnione z klasą

```
class Wektor
{
private:
    double x, y;
public:
    // Konstruktor
    Wektor(double x = 0.0, double y = 0.0) : x(x), y(y) {}
    void wypisz() const
    {
        cout << "[" << x << ", " << y << "]\n";
    }
 friend Wektor operator+(const Wektor &w1, const Wektor &w2);
};
```

W takim wypadku najlepszym sposobem jest zaprzyjaźnienie metody z klasą.

```
Wektor operator+(const Wektor &w1, const Wektor &w2)
{
    return Wektor(w1.x + w2.x, w1.y + w2.y);
}
```

```
class wektor
{
    private:
        int x;
        int y;
    public:
        wektor(int a = 0, int b = 0): x(a), y(b) {}
        friend wektor operator+(wektor w1, wektor w2);
        friend wektor operator-(wektor w1, wektor w2);
        friend wektor operator-(wektor w);
        friend wektor operator*(wektor w, int skalar);
        friend wektor operator*(int skalar, wektor w);
        friend void operator+=(wektor& w1, wektor w2);
        friend void operator+=(wektor& w1, int skalar);
        friend void operator*=(wektor& w, int skalar);
        friend ostream& operator<<(ostream& stm, wektor w);
        friend istream& operator>>(istream& stm, wektor& w);
};
wektor operator+(wektor w1, wektor w2)
{
wektor operator-(wektor w1, wektor w2)
{
wektor operator-(wektor w)
{
wektor operator*(wektor w, int skalar)
{
wektor operator*(int skalar, wektor w)
{
void operator+=(wektor& w1, wektor w2)
{
void operator+=(wektor& w1, int skalar)
{
void operator*=(wektor& w, int skalar)
{
ostream& operator<<(ostream& stm, wektor w)
{
istream& operator>>(istream& stm, wektor& w)
{
int main()
{
```

Funkcje operatorowe zaprzyjaźnione z klasą

Zadeklarowana w klasie funkcja zaprzyjaźniona nie musi być zaimplementowana w programie (to nie błąd).

Tak więc, projektując klasę można „na zapas” zaprzyjaźnić ją ze wszystkimi operatorami.

Przeciążanie operatorów

3. Funkcje operatorowe zdefiniowane jako
   metody klasy

Operatory zdefiniowane jako metody

Funkcje operatorowe zdefiniowane jako metoda klasy.

Funkcja operatorowa zdefiniowana wewnątrz klasy przyjmuje o jeden argument mniej, niż zapisana jako funkcja globalna.

Domyślnie, jej pierwszym (brakującym) argumentem jest this – czyli obiekt klasy na rzecz której została wywołana

```
class klasa
{
    klasa operator+ (klasa);
};

klasa klasa::operator+(klasa b)
{
    //treść metody
}
```

Domyślnie - (this, klasa b)

- W tym przypadku funkcja operatorowa musi być niestatyczną metodą klasy, dla której pracuje.

Operatory zdefiniowane jako metody

```
class wektor
{
    private:
        int x;
        int y;
    public:
            wektor(int a = 0, int b = 0): x(a), y(b) {}
            wektor operator+(wektor w2);
            wektor operator-(wektor w2);
            wektor operator-();
            wektor operator*(int skalar);
        friend wektor operator*(int skalar, wektor w);
            void operator+=(wektor w2);
            void operator+=(int skalar);
            void operator*=(int skalar);
        friend ostream& operator<<(ostream& stm, wektor w);
        friend istream& operator>>(istream& stm, wektor& w);
};
wektor wektor::operator+(wektor w2)
{
    wektor wynik(this->x + w2.x, this->y + w2.y);
    return wynik;
}
wektor wektor::operator-()
{
    wektor wynik(-x,   -y);
    return wynik;
}
```

Operatora nie mona zdefiniować jako metody klasy, gdy pierwszym jego argumentem (tym domyślnym) NIE JEST obiekt na rzecz którego go wykonujemy.

Przeciążanie operatorów

Operator przypisania

Operator przypisania „ = ”
musi być zdefiniowany jako metoda klasy

Operator przypisania

Znak „ = ” może, w zależności od kontekstu, wywołać **funkcję operatora przypisania** lub **konstruktora kopiującego**.

Znak „ = ” uruchamia konstruktor kopiujący gdy wystąpi on w linii definicji obiektu. Symbol ten wtedy oznacza inicjalizację, a nie przypisanie.

- Inicjalizacją zajmuje się konstruktor kopiujący.
- Przypisaniem zajmuje się operator przypisania.

Jeśli operator nie zostanie zdefiniowany – zostanie automatycznie wygenerowany przez kompilator, a przypisanie odbędzie się metodą „składnik po składniku”, podobnie jak w przypadku konstruktora kopiującego.

Operator przypisania

```
class osoba
{
public:
     string * imie;
     osoba(string kto)
     {
          imie = new string;
          *imie = kto;
     }
     osoba (osoba &);
     ~osoba() {delete imie;}
     osoba operator=(osoba wzorzec)
     {
         imie = new string;
         *imie = *wzorzec.imie;
     }
};
```

Definiowanie własnego operatora przypisania ma sens głównie w przypadku klas posiadających wskaźniki do obiektów tworzonych dynamicznie.

Patrz wykład o konstruktorze kopiującym.

Operator przypisania

Operator przypisania **nie jest** automatycznie generowany gdy:

- klasa ma pole typu ***const*** - pole typu ***cons***t może być jedynie inicjalizowane nie wolno nic do niego przypisywać.
- klasa ma pole będące referencją.
- klasa ma pole będące obiektem innej klasy i w tej innej klasie operator przypisania jest prywatny – wymagane jest użycie operatora przypisania klasy obiektu-składnika.
- klasa ma klasę podstawową, w której operator przypisania jest w sekcji prywatnej.

Przeciążanie operatorów

Operator porównania

Operator porównania „ == ”
musi być zdefiniowany jako metoda klasy

Przeciążanie operatorów

```
class Wektor
{
protected:
    double x;
    double y;

public:
    wektor(double X = 0, double Y = 0) : x(X), y(Y) {}
    friend bool operator==(Wektor w1, Wektor w2);
};

bool operator==(Wektor w1, Wektor w2)
{
    if (w1.x == w2.x && w1.y == w2.y)
        return true;
    else
        return false;
}
```

Przeciążanie operatorów

```
class Wektor
{
protected:
    double x;
    double y;

public:
    wektor(double X = 0, double Y = 0) : x(X), y(Y) {}
    bool operator==(Wektor w2)
    {
        if (x == w2.x && y == w2.y)
            return true;
        else
            return false;
    }
};
```

Przeciążanie operatorów

Operatory inkrementacji (++)

i dekrementacji (--)

Operatory post- i pre- inkrementacji, oraz dekrementacji

Przeciążanie operatorów

```
class liczba {
  int x;
public:
  liczba(int y): x(y) {}

  // preinkrementacja - najpierw  zwiększamy, a potem zwracamy wartość
  liczba & operator++() {
    ++x;
    return *this;
  }

  // postinkrementacja - najpierw zwracamy wartość, a potem zwiększamy
  liczba operator++(int) {  // specjalny zapis do postinkrementacji
    liczba kopia = (*this);
    ++x;  // zwiększa liczbę przechowywaną w orginale (kopia zostaje bez zmian)
    return kopia; // zwracamy kopię, a nie oryginał
  }
};
```

Przeciążanie operatorów

Operator [ ]

Operator [ ]

```
class tablica
{
public:
    int n;
    int *tab;
    tablica(int ile)
    {
        n=ile;
        tab = new int[n];
    }
    ~tablica(){delete [] tab;}
    int& operator[](unsigned int index)
    {
        return tab[index];
    }

};

int main()
{
    tablica t1(10);
    t1[0] = 100;
    return 0;
}
```

Dzięki przeciążeniu operatora [ ] z klasy zawierającej tablicę można korzystać jak ze zwykłej tablicy.

Literatura:

W prezentacji wykorzystano przykłady i fragmenty:

- Gręboszj J. : ***Symfonia C++, Programowanie w języku C++ orientowane obiektowo***, Wydawnictwo Edition 2000.
- Jakubczyk K.: *Turbo Pascal i Borland C++ Przykłady*, Helion.

Warto zajrzeć także do:

- Sokół R. : ***Microsoft Visual Studio 2012 Programowanie w Ci C++,*** Helion.
- Kerninghan B. W., Ritchie D. M.: ***język ANSI C***, Wydawnictwo Naukowo Techniczne.

Dla bardziej zaawansowanych:

- Gręboszj J. : ***Pasja C++,*** Wydawnictwo Edition 2000.
- Meyers S.: ***język C++ bardziej efektywnie***, Wydawnictwo Naukowo Techniczne