PRZECIĄŻANIE OPERATORÓW

235

C++ - przeciążanie operatorów

Operatory są elementami języka C++.

Istnieje zasada, że z elementami języka, takimi jak np. słowa kluczowe, nie można dokonywać żadnych zmian, przeciążeń, itp.

Ale dla operatorów zrobiono wyjątek:

operatory można przeciążać.

236

C++ - przeciążanie operatorów

Co to znaczy – *przeciążyć operator*?

Przeciążanie operatorów polega na napisaniu nowych funkcji lub metod, które wywołujemy w taki sam sposób jak operatory.

Nie da się modyfikować sposobu działania już zdefiniowanych przypadków użycia danego operatora, ale można wprowadza nowe, których do tej pory nie było i których obecność ułatwi pisanie kodu.

237

C++ - przeciążanie operatorów

Nie przeciąża się operatora dodawania dwóch liczb typu **`double`**. Można natomiast przeciążyć operator dodawania tak aby można było pisać wyrażenia arytmetyczne, których argumentami są obiekty zdefiniowanego przez nas typu.

Dlatego..

jedynie wyrażenia obejmujące typy zdefiniowane przez użytkownika mogą zawierać przeciążone operatory.

238

C++ - przeciążanie operatorów

Definiując funkcję reprezentującą przeciążony operator, jako jej nazwę piszemy: **`operator@`**, gdzie znak `@` reprezentuje przeciążony operator (np. +, -, /, *..)

Istnieją cztery możliwości:

1. Operator przeciążony może być zdefiniowany jako
   1. **funkcja globalna**, albo
   2. **metoda** należąca do klasy.
2. Operator przeciążony może być
   1. **jedno-** , albo
   2. **dwu**argumentowy.

239

C++ - przeciążanie operatorów

Jeżeli przeciążamy operator, który ma działać jako funkcja globalna, to po słowie **`operator@`** zapisujemy

1. **dwa argumenty** jeżeli jest to operator dwuargumentowy, lub
2. **jeden** – jeżeli jednoargumentowy

Jeżeli przeciążamy operator, który ma działać jako metoda należąca do klasy, to po słowie **`operator@`** mamy

1. **brak argumentów**, jeżeli operator jest jednoargumentowy, lub
2. **jeden argument** – jeżeli dwuargumentowy. W tym przypadku obiekt, na rzecz którego będzie wywoływany przeciążony operator staje się argumentem *lewostronnym*

240

C++ - przeciążanie operatorów

Przykład:

Zdefiniujmy klasę, która ma reprezentować liczbę typu całkowitego (integer).

Chcielibyśmy, aby na obiektach tej klasy można było dokonywać operacji arytmetycznych w taki sam sposób, jak na zmiennych typu **`int`**.

C++ - przeciążanie operatorów

Klasa będzie nazywała się `Integer`

Na początek będziemy dla niej potrzebowali:

1. konstruktora pozwalającego na ustalenie przechowywanej wartości oraz
2. operator dwuargumentowy: += , przy czym argumentami operatora mają być obiekty typu `Integer`

```
class Integer {
    int i;
public:
  Integer(int ii): i(ii) {}
  Integer& operator+=(const Integer& rv) {
        i += rv.i;
        return *this;
  }
  int get_i() { return i; }
};

int main(int argc, char *argv[])
{
  Integer I(1), J(2);
  I += J;
```


C++ - przeciążanie operatorów

Ponieważ operator jest dwuargumentowy, definiujemy jeden argument wywołania

Przekazujemy argument przez referencje aby niepotrzebnie nie wymuszać tworzenia kopii

Deklarujemy argument jako **`const`** aby zagwarantować mu „nietykalność"

Ponieważ operator nie zwraca żadnej wartości, moglibyśmy napisać:

```
void operator+=(const Integer& rv)
```

ale..

```
class Integer {
    int i;
public:
  Integer(int ii): i(ii) {}
  Integer& operator+=(const Integer& rv) {
        i += rv.i;
        return *this;
  }
  int get_i() { return i; }
};

int main(int argc, char *argv[])
{
  Integer I(1), J(2);
  I += J;
```


C++ - przeciążanie operatorów

..w C++ przyjmujemy że wszystkie operacje przypisania oprócz czynności przekopiowania wartości dodatkowo zwracają je jako rezultat swego działania.

To umożliwia tworzenie wyrażeń kaskadowych:

A+=B+=C+=D;

Aby zachować tę właściwość deklarujemy, że operator zwraca referencję do obiektu typu `Integer`

```
class Integer {
    int i;
public:
  Integer(int ii): i(ii) {}
  Integer& operator+=(const Integer& rv) {
        i += rv.i;
        return *this;
  }
  int get_i() { return i; }
};

int main(int argc, char *argv[])
{
  Integer I(1), J(2);
  I += J;
```


C++ - przeciążanie operatorów

Operator dodawania

Działając na typach wbudowanych, dodawanie polega na utworzeniu zmiennej pomocniczej i zapisaniu do niej wyniku sumowania wartości z lewego i prawego argumentu wyrażenia:

```
int a, b=1, c=2;
a = b + c;
```


C++ - przeciążanie operatorów

Operator dodawania

```
const Integer operator+(const Integer& rv) {
            return Integer(i+rv.i);
}
```

Metoda tworzy zmienną lokalną takiego samego typu, której zadaniem będzie przechować wynik dodawania.

Metoda zwraca tę zmienną, która jest następnie kopiowana np. do zmiennej po lewej stronie równości: **a = b + c;**

Po kopiowaniu zmienna lokalna jest usuwana.

C++ - przeciążanie operatorów

```
const Integer operator+(const Integer& rv) {
                return Integer(i+rv.i);
  }
```

Zmienna lokalna zwracana przez metodę jest obiektem tymczasowym: ma za zadanie tylko zwrócić wartość i zniknąć. Nie zwracamy jej przez podanie jej adresu (*to byłoby niebezpieczne!*), tylko przez wartość

Ponadto deklarujemy ją jako **`const`**. Ma to chronić przed takimi „wybrykami", jak np. wywołanie metody należącej do obiektu zwracanego przez wyrażenie, np. : **`(a+b).fun()`**

Dzięki uczynieniu wartości zwracanej jako **`const`** określa się, że dla tej wartości mogą być wywołane jedynie stałe składowe (również metody) klasy **`Integer`**. To zapobiega umieszczeniu w obiekcie nowej, potencjalnie wartościowej informacji

C++ - przeciążanie operatorów

```
class Integer {
    int i;
public:
    Integer(int ii): i(ii) {printf("%i\n",i); }
    Integer& operator+=(const Integer& rv) {
          i += rv.i;
          return *this;
    }
    const Integer operator+(const Integer& rv) {
          return Integer(i+rv.i);
    }
    int get_i() { return i; }
};
```

```
int main(int argc, char *argv[])
{
   Integer I(1), J(2), K(3);
   I += J + K;
   printf("%i\n",I.get_i());
   …
   // co pojawi się w oknie konsoli?
```

Visual Studio

C++ - przeciążanie operatorów

Czego nie wolno robić?

- Nie można tworzyć operatorów, które nie posiadają znaczenia w języku C, np. poprzez łączenie już istniejących (nowy operator **`**`** utworzony w ten sposób mógłby oznaczać potęgowanie, ale niestety..)
- Nie wolno zmieniać kolejności obliczania operatorów
- Nie wolno zmieniać liczby argumentów

C++ - przeciążanie operatorów

Operatory jednoargumentowe minus i negacja:

```
class Integer {
    int i;
public:
const Integer operator-() {
        return Integer(-i);
    }
 const Integer operator!() {
        return Integer(!i);
    }
…
Integer I(1), J(2), K(3);
 I = -J;  printf("%i\n",I.get_i());
 I = !K; printf("%i\n",I.get_i());
```


C++ - przeciążanie operatorów

Inkrementacja i dekrementacja

Jak rozróżnić wywołanie przyrostkowe i przedrostkowe (*operator w obydwu przypadkach wygląda tak samo*)?

Okazuje się, że kompilator już sobie z tym poradził pierwszy w związku z typami wbudowanymi:

1. jeżeli kompilator widzi **`++a`**, generuje wywołanie funkcji **`operator++(a)`**
2. jeżeli kompilator widzi **`a++`**, generuje wywołanie funkcji **`operator++(a,int)`**

Kompilator rozróżnia te dwie postacie wywołując dwie funkcje mające różne sygnatury. Kompilator sam przekazuje sztuczną, stałą wartość argumentu typu **`int`**

C++ - przeciążanie operatorów

```
class Integer {
    int i;
public:
 const Integer& operator++() {
        i++;
        return *this;
    }
 const Integer operator++(int) {
        Integer wart_przed(i);
        i++;
        return wart_przed;
    }
```

```
class Integer {
  int i;
public:
…
  friend const Integer& operator--(Integer &a);
  friend const Integer operator--(Integer &a, int);
};
const Integer& operator--(Integer &a) {
        a.i--;
        return a;
}
const Integer operator--(Integer &a, int) {
        Integer wart_przed(a.i);
        a.i--;
        return wart_przed;
}
```

C++ - przeciążanie operatorów

```
class Integer {
    int i;
public:
 const Integer operator*(
                    Integer &a) {
      return Integer(i *a.i);
   }
```

```
class Integer {
 int i;
public:
...
 friend const Integer operator/(Integer &arg_lewy,
                                Integer &arg_prawy);
};
const Integer operator/(Integer &arg_lewy,
                                Integer &arg_prawy) {
      if (arg_prawy.i == 0)
        return Integer(INT_MAX);
      else
        return Integer(arg_lewy.i/arg_prawy.i);
}
```


253

C++ - przeciążanie operatorów

Argumenty wywołania:

1. jeżeli w kodzie operatora zamierzamy tylko odczytywać wartość przekazaną w argumentach, argument powinien zostać przekazany jako referencja do stałej (`const`).
2. Jedynie w przypadku operatorów zmieniających argument po lewej stronie (połączonych z przypisaniem, jak np. `+=` oraz `operator=`) argument po lewej stronie nie może być `const`.

Zwracane wartości:

1. Typ wartości zwracanej przez operator powinien zależeć od spodziewanego znaczenia operatora.
2. Wartości zwracane przez operatory przypisania powinny być referencjami do l-wartości, niebędącymi referencjami do stałych.
3. Wynikiem działania operatorów logicznych są przynajmniej liczby całkowite a najlepiej – wartości typu `bool`. Nie należy tego zmieniać.

254

C++ - przeciążanie operatorów

W przeciążonym operatorze mnożenia można napisać tak:

```
const Integer operator*(Integer &a) {
    return Integer(i *a.i);
 }
```

albo tak:

```
const Integer operator*(Integer &a) {
     Integer tmp(i * a.i);
     return tmp;
 }
```

Czy jest różnica w działaniu?

255

C++ - przeciążanie operatorów

W przeciążonym operatorze mnożenia można napisać tak:

```
return Integer(i * a.i);
```

albo tak:

```
Integer tmp(i * a.i);
return tmp;
```

Przypadek pierwszy:

utwórz tymczasowy obiekt w miejscu przeznaczonym na wartość zwracaną przez funkcję – ***jedna operacja***

Przypadek drugi:

utwórz lokalny obiekt 'tmp', wywołaj konstruktor kopiujący, żeby skopiować 'tmp' do miejsca znajdującego się na zewnątrz funkcji, przeznaczonego na zwracaną przez nią wartość; wywołaj destruktor 'tmp' usuwając ten obiekt – ***trzy operacje***

256

C++ - przeciążanie operatorów

Przeciążanie operacji przypisania

Gdy używany jest znak '=' to czasem następuje przypisanie, a czasem – wywoływany jest konstruktor kopiujący, np. :

```
1. Integer a(1);
2. Integer b = a;
3. b = a;
```

W drugiej linijce tworzony jest nowy obiekt oraz inicjalizowany wartością 'a'. Przy tworzeniu wywoływany jest zawsze konstruktor. Kompilator połączy to w całość i wywoła konstruktor kopiujący: b(a)

W trzeciej linijce zostanie wywołana metoda

```
Integer::operator=
```


257

C++ - przeciążanie operatorów

Z reguły przy inicjalizowaniu obiektu za pomocą '=', zamiast dwóch operacji: (1) wywołania konstruktora domyślnego i (2) operacji przypisania, kompilator chce wykonać jedną: poszukuje konstruktora jednoargumentowego, który przyjmie jako argument to, co znajduje się po prawej stronie operacji przypisania.

Dlatego dla jasności lepiej jest unikać wykorzystywania znaku '=' do inicjalizacji obiektów i zamiast tego zawsze używać wywołania konstruktora

258

C++ - przeciążanie operatorów

A jeżeli mamy zwykłe przypisanie wartości..?

Jeżeli programista nie zdefiniuje operatora przypisania to kompilator dostarczy własny **operator przypisania**, który skopiuje obiekt pole po polu.

Ten operator może generować błędy, ponieważ:

- nie będzie działał dla pól stałych,
- nie będzie działał dla referencji,
- dla wskaźników skopiuje wartość.

C++ - przeciążanie operatorów

Własny operator przypisania

```
class Integer {
      int i;
public:
  const Integer operator=(const Integer &rv) {
      i = rv.i;
            return *this;
       }
 …
```

Gdyby wśród składowych klasy były wskaźniki, należałoby rozstrzygnąć, czy je przepisywać, czy kopiować ich zawartość, ewentualnie – jeżeli rozmiary struktur nie byłyby zgodne – usunąć poprzednie i założyć nowe o nowym rozmiarze.

C++ - przeciążanie operatorów

Przeciążanie operatorów []

Przykład:

klasa reprezentująca typ tablicowy. Obiekt ma reprezentować tablicę, do której można się odwoływać intuicyjnie, np. `Tab[i]`

Ma być też dostępnych kilka innych metod ułatwiających operacje na tej tablicy, np. `rozmiar()`.

C++ - przeciążanie operatorów

Przykład:

```
class Tablica {
   int rozm;
   int *wsk;
public:
   Tablica( int = 10 );
   Tablica( const Tablica& );
   ~Tablica();
   int rozmiar(void) const {return rozm; }
   const Tablica &operator=( const Tablica& );
   bool operator==( const Tablica& );
   int &operator[]( int );
   const int &operator[]( int ) const;
};
```


C++ - przeciążanie operatorów

Implementacja przeciążonego operatora []:

```
int &Tablica::operator[]( int i) {
    assert (0 <= i && i < rozm);
    return wsk[i];
}

// operator dla obiektów zadeklarowanych jako const
const int &Tablica::operator[]( int i) const {
    assert (0 <= i && i < rozm);
    return wsk[i];
}
```


C++ - przeciążanie operatorów

Przykład wykorzystania:

```
int main(int argc, char *argv[])
{
    Tablica Tab1(5), Tab2;
    for (int i=0; i<Tab1.rozmiar(); i++)
        Tab1[i] = i;
    Tab2 = Tab1;
    assert(Tab2 == Tab1);
    …
```

Visual Studio

C++ - przeciążanie operatorów

Przeciążanie a dziedziczenie

Operatory, *z wyjątkiem operatora przypisania* są automatycznie dziedziczone w klasach pochodnych.

```
class Integer {
    int i;
public:
    Integer(int ii): i(ii) {}
    Integer& operator+=(const Integer& rv) {
        i += rv.i;
        return *this;
    }
     Integer& operator=(const Integer &rv) {
        i = rv.i;
        return *this;
    }
    Integer& operator=(int ri) {
        i = ri;
        return *this;
    }
};
```

```
class Integer2: public Integer {
public:
    Integer2(int i): Integer(i) {}
    Integer2& operator=(const Integer& prawy) {
        Integer::operator=(prawy);
        return *this;
    }
    Integer2& operator=(int ri) {
        Integer::operator=(ri);
        return *this;
    }
};
int main(int argc, char *argv[])
{
    Integer2 I2(1), J2(0);
    Integer K1(9);
    J2 = 5;   // metoda własna
    I2 = K1;  // metoda własna
    J2 = I2;  // ? – żaden z istniejących..
    I2 += J2; // metoda dziedziczona
```


C++ - przeciążanie operatorów

Automatyczne tworzenie operatora =

Ponieważ programiści oczekują, że operacja przypisania dla dwóch zmiennych jednakowego typu zawsze powinna się powieść, dlatego kompilator zawsze automatycznie tworzy ten operator, jeżeli programista go nie zdefiniuje.

Działanie tego operatora jest identyczne jak domyślnego konstruktora kopiującego:

jeżeli klasa zawiera składowe obiekty, to dla każdego z nich wywoływany jest rekurencyjnie operator = (tzw. *przypisanie za pośrednictwem elementów składowych*).

Nie należy na to pozwalać kompilatorowi, ale samemu pisać operator przypisania, jeżeli planuje się go używać – zdanie się na automatycznie wygenerowane operatory łatwo prowadzi do błędów.