Instytut Informatyki UMCS
Zakład Technologii Informatycznych

Wykład 5

Operatory

dr Marcin Denkowski

Lublin, 2019

AGENDA

1. Przeciążanie operatorów
2. Przegląd operatorów
3. Zaprzyjaźnianie
4. Operator przypisania (=)
5. Operatory rzutowania
6. Operator strumienia
7. Operatory alokacji i dealokacji

PRZECIĄŻANIE OPERATORÓW

Przeciążanie operatorów (*operator overloading*)

- możliwość tworzenia funkcji operatorowych z typami złożonymi, w postaci

1) `operator op`
2) `operator type`
3) `operator new {new []}`
4) `operator delete {delete []}`

op – jeden z operatorów:

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

OPERATORY – WŁASNOŚCI

- Restrykcje:
  - nie można tworzyć własnych operatorów (np. **, <>)
  - nie można zmienić priorytetu istniejących operatorów
  - niektóre operatory mogą występować tylko jako metody klasy
  - przynajmniej jeden z operandów musi być typem złożonym
  - nie można zmienić ilości operandów dla danego operatora
- Zasady
  - nie ma obostrzeń co do zwracanego typu
  - z zasady przeciążone operatory powinny zachowywać się podobnie do operatorów wbudowanych (kanoniczne zachowanie)

OPERATORY, ZESTAWIENIE

Przypisania	Inkrementacji/ dekrementacji	Arytmety-czne	Logiczne	Porównania	Dostępu	Inne
a = b a += b a -= b a *= b a /= b a %= b a &= b a |= b a ^= b a <<= b a >>= b	++a --a a++ a--	+a -a a + b a - b a * b a / b a % b ~a a & b a | b a ^ b a << b a >> b	!a a && b a || b	a == b a != b a < b a > b a <= b a >= b	a[b] *a &a a->b *a.b* *a->*b* *a.*b* *a::b*	a(...) a, b (type)a *?:* *sizeof* new delete **_cast*

PRIORYTET OPERATORÓW

Priorytet	Operator	Opis	Łączność
1	a++, a--, (), [], ., →, (typ)	Funkcje, tablice, dostępu	Od lewej
2	++a, --a, +a, -a, !, ~, *(dereferencja), &(adres), sizeof, new, delete	Rzutowania, pamięciowe, arytmetyczne unarne	Od prawej
3	a*b, a/b, a%b	Arytmetyczne binarne	Od lewej
4	a+b, a-b	Arytmetyczne binarne	Od lewej
5	<< >>	Przesunięcia bitowe	Od lewej
6	<, >, <=, >=	Relacji <, >	Od lewej
7	==, !=	Relacji ==	Od lewej
8	&, ^, |, &&, ||	Bitowe i logiczne	Od lewej
9	? :	Warunkowe	Od prawej
10	=, +=, -=, ...	przypisania	Od prawej
11	,	przecinek	Od lewej

PRZECIĄŻANIE OPERATORÓW

Przeciążanie operatorów symbolicznych

Wyrażenie	Jako metoda	Jako funkcja	Przykład
@a	(a).operator@()	operator@(a)	!obj ↔ obj.operator!()
a@b	(a).operator@(b)	operator@(a,b)	obj+5 ↔ obj.operator+(5)
a=b	(a).operator=(b)	niemożliwy	obj = obj2 ↔ obj.operator=(obj2)
a(b...)	(a).operator()(b...)	niemożliwy	Object obj; obj(b) ↔ obj.operator()(b)
a[b]	(a).operator[](b)	niemożliwy	obj[b] ↔ obj.operator[](b)
a->	(a).operator->()	niemożliwy	ptr_obj->member ↔ obj.operator->
a@	(a).operator@(0)	operator@(a, 0)	obj++, obj.operator++(0)

OPERATORY ARYTMETYCZNE

- Zazwyczaj jako funkcje globalne
- Część operatorów jest zarówno unarna jak i binarna

```
class X {
public:
     X& operator+=(const X& rhs) {
          // instrukcje ...
          return *this;
     }

     X& operator++() {  //pre-inkrementacja
          // instrukcje ...
        return *this;
     }

     X operator++(int) {  //post-inkrementacja
          X tmp(*this);  // kopia oryginalu
          operator++();  // pre-inkrementacja
          return tmp;    // return oryginal
     }

     friend X operator+(X l, const X& r);
};
```

```
X operator+(X l, const X& r) {
  l += r;
  return l;
}
```

OPERATORY RELACJI

- *Relational operators*
- Używane zwykle przy algorytmach wymagających porównania
- W postaci kanonicznej zwraca **`bool`**

```
bool operator< (const X& l, const X& r) { /*instrukcje porownania*/ }
inline bool operator> (const X& l, const X& r){ return r < l; }
inline bool operator<=(const X& l, const X& r){ return !(l > r); }
inline bool operator>=(const X& l, const X& r){ return !(l < r); }

inline bool operator==(const X& l, const X& r){ /*instrukcje porownania*/ }
inline bool operator!=(const X& l, const X& r){ return !(l==r); }
```

OPERATOR PRZYPISANIA

- *Assignment operator*
- Musi być nie-statyczną funkcją składową klasy
- Jest wywoływany na rzecz nowej klasy, w parametrze dostaje obiekt klasy, która jest źródłowa
- W postaci kanonicznej zwraca referencję na **`this`**, a w parametrze bierze stałą referencję na inny obiekt

```
class T
{
    T& operator=(const T& other)
    {
        if (this != &other) // sprawdzenie tozsamosci
        {
            // instrukcje kopiujace skladowe
        }
        return *this;
    }
};
```

OPERATOR RZUTOWANIA

- Funkcja konwersji (*user-defined conversion*)

```
operator type()
```

- Jest używana podczas jawnej lub niejawnej konwersji typu
- Musi być nie-statyczną funkcją składową klasy
- Metoda nie zwraca żadnego typu/wartości
- Może prowadzić do wieloznaczności z konstruktorem konwertującym

```
class D;
class T {
    operator int() const { return 7; }
    operator bool() const { return v; }

    [explicite] T(const D& ) {…}          // konstruktor konwertujacy
    operator D() const { return D; }      // operator konwersji
};
…
T t;
if (t) {
    D d(t); // konstruktor
    d = t;  // operator D() – niejawna konwersja
}
```

OPERATOR STRUMIENIA

- *Stream extraction/insertion operators*
- Klasy strumieniowe **`ostream`** i **`istream`** z biblioteki **`iostream`**
- Kanoniczny operator wstawiania do strumienia

  `operator<<(ostream&, const T&)`
- Kanoniczny operator wyjmowania ze strumienia

  `operator>>(istream&, T&)`
- Funkcje globalne

```
ostream& operator<<(ostream& os, const T& obj) {
    // os << …; // zapis do strumienia
    return os;
}

istream& operator>>(istream& is, T& obj) {
    // is >> …; // odczyt ze strumienia
    return is;
}
…
cout << 5 << „napis” << endl;
```

OPERATOR TABLICOWY

- *Array subscriptor operator*
- Zapewnia dostęp do składowej na sposób tablicowy
- Kanonicznie zwraca referencję na obiekt, a w parametrze dostaje obiekt indeksujący

```
class T
{
    S& operator[](int idx) {
        return element[idx];
    }

    S& operator[](string name) {
        if(name == test) return element;
    }
};
…
T t;
S s = t[3];
s   = t[„elem”];
```

FUNKTOR

- *Function call operator*
- Klasa, która przeciąża operator wywołania funkcji () staje się *funktorem*
- Obiekt takiej klasy może być użyty jak funkcja
- Wiele algorytmów biblioteki standardowej używa takich klas

```
struct Sum
{
    int sum;
    Sum() : sum(0) { }
    void operator()(int n) { sum += n; }
};

int array[] = {1,2,3,4,5,…};
Sum s = std::for_each(array, array+n, Sum());
```