Programowanie obiektowe

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja:16.12.2019

4. Przeciążanie funkcji i operatorów

Przeciążanie funkcji i operatorów

W języku C wymagane jest stosowanie unikalnych nazw funkcji.

Przykład

- Operacje `wash car` i `wash face` w języku C muszą być wyrażone jako `wash_car(struct car*)` oraz `wash_face(struct face*)`.
- W języku C++ możemy użyć tej samej nazwy w różnych kontekstach:

```
wash(struct car*)
wash(struct face*)
class Car{
   public: void wash();
};
```

Przeciążanie funkcji i operatorów

Interpretując wywołanie funkcji, kompilator wybierze wywołanie odpowiedniej implementacji na podstawie argumentów występujących w wywołaniu.

```
double max(double d1, double d2){
    return (d1>d2 ? d1:d2);
}
double max(int d1, int d2){
    return (d1>d2 ? d1:d2);
}

max(1.0,2.0) ; // wywoła wersję double
max(1,2) ; // wywoła wersję int
```

Rozróżnienie typów argumentów

- W trakcie wywołania przeładowanych funkcji kompilator wybiera wersję funkcji najlepiej pasującą do typu argumentów. Jeżeli odpowiednia funkcja zostanie odnaleziona, wówczas jest ona wołana.
- W przeciwnym przypadku kompilator będzie raportował niejednoznaczność traktowaną jako błąd.

```
// nieodróżnialne od double max(double d1, double d2)
double max(const double&d1, const double&d2){
    return (d1>d2 ? d1:d2);
}

// możliwe są dwie ścieżki automatycznej konwersji
max(1.0,2);
```

Dopasowania i konwersje

- **dokładne dopasowanie** argumentów wywołania do jednej z definicji
- następuje **trywialna konwersja**

`type-name`	`type-name&`
`type-name&`	`type-name`
`type-name[]`	`type-name*`
`type-name`	`const type-name`
`type-name*`	`const type-name*`

- następuje **konwersja całkowitoliczbowa** pomiędzy danymi typu `int, long, unsigned`
- istnieje **standardowa konwersja** pomiędzy argumentami

`void*`	`const void*`
`DerivedClass*`	`BaseClass*`
`DerivedClass&`	`BaseClass&`

- istnieją **zdefiniowane przez użytkownika** konwersje

`String`	`operator const char*()`
`const char*`	`String`

- w definicji funkcji pojawia się **elipsa** ...

Standardowe argumenty

Alternatywą do implementacji kilku przeciążonych wersji funkcji o podobnym zachowaniu jest zadeklarowanie jej standardowych argumentów.

```
int print(char *s ); // (1) Print a string.
// Print a double with default precision
int print(double dvalue); // (2)
// Print a double with a given precision.
int print(double dvalue, int prec);  // (3)
```

Funkcje (2)oraz (3) są bardzo do siebie podobne, funkcja (2) może być zaimplementowana jako

```
int print(double dvalue)
{
    return print(dvalue,DEFAULT_PRECISION);
}
```

Standardowe argumenty

Analogiczny efekt, jaki daje przeciążenie uzyskamy rezygnując z implementacji funkcji (2) oraz deklarując funkcję (3) jako:

```
int print(double dvalue, int prec = DEFAULT_PRECISION);
```

- Kompilator napotykając wywołanie: `print(4.5,3)` wywoła funkcję (3) z argumentem `prec = 3`.
- Napotykając wywołanie `print(5.1)` automatycznie wygeneruje wywołanie `print(5.1, DEFAULT_PRECISION)`.

Przeciążanie operatorów

Przeciążanie operatorów w C++ jest zabiegiem wyłącznie syntaktycznym. Wszystkie operatory są implementowane jako funkcje. Różnicą jest postać wywołania.

Zamiast pisać

```
x5 = plus(plus(plus(x1,x2),x3),x4) ;
```

możemy użyć zapisu

```
x5= x1+x2+x3+x4 ;
```

- Podobnie, jak w przypadku przeciążonych funkcji, kompilator automatycznie dobiera odpowiedni operator dokonując, jeżeli jest to wymagane, automatycznych konwersji.
- Nie można redefiniować trójargumenowego operatora warunkowego wyboru oraz kilu innych (. :: .*).

Składnia

- Operatory w C++ definiuje się z użyciem słowa kluczowego operator, po którym następuje nazwa operatora. Operatory mogą być składowymi klas lub mogą być zadeklarowane jako funkcje globalne.
- Liczba argumentów operatora uzależniona jest od jego typu oraz miejsca definicji.
- W C/C++ występują operatory:
  - Unarne (jednoargumentowe)
  - Binarne (dwuargumentowe)
  - Jeden operator terenarny (trójargumentowy) ?

```
z = x<0 ? 0 : x;
```

	Unarny	Binarny
Globalny	1 argument	2 argumenty
Lokalny (metoda klasy)	0 argumentów	1 argument

Składnia

- Składnia wywołania przeciążonych operatorów jest zgodna ze składnią operatorów C/C++ dla wbudowanych typów.
- Operatory nie mogą mieć standardowych argumentów
- Wszystkie przeciążone operatory (poza operatorem przypisania `operator=`) są dziedziczone.
- Podczas wywołania operatorów pierwszym argumentem musi być zawsze typ (referencja typu), dla której operator został zdefiniowany. **Kompilator nigdy nie stosuje konwersji dla pierwszego argumentu**.

Przykład

```
class String
{
public:
    char buf[256];
    String(const char*txt=""){strcpy(buf,txt);}
    operator const char*()const {return buf;}
    String&operator=(const String&s)   {
        strcpy(buf,s.buf);
        return *this;
    }
    char&operator[](int idx){
        if(idx<0 || idx>255)return buf[0];//throw 0;
        return buf[idx];
    }
};
```

Przykład

```
String&operator+=(String&s,const char*txt){
    strcat(s.buf,txt);
    return s;
}

const String operator+(const String&s,const char*txt) {
    String r(s.buf);
    strcat(r.buf,txt);
    return r;
}

bool operator==(const String&s1,const char*txt)
{
    return !strcmp(s1.buf,txt);
}
```

Przykład

```
int main(){
    String a("Ala ma");
    a+=" ";
    String b("kota");
    a+=b;  // automatycznie wywoła
    //String::operator const char*()
    if(a=="Ala ma kota"){…}
    b = b + " i psa";
    for(int i=0;i<strlen(b);i++)putchar(b[i]);
}
```

Operatory inkrementacji i dekrementacji

- Unarne (jednoargumentowe) operatory inkrementacji i dekrementacji występują w dwóch odmianach: prefiksowej i postfiksowej;
- Typowa implementacja jest następująca:

```
class Int{
    int value;
public:
    Int&operator++(){
        value++;return *this;
    }
    Int operator++(int){
        Int temp = *this;
        ++*this;
        return temp;
    }
};
```

Operator przypisania

- Operator przypisania operator= musi być zadeklarowany jako metoda klasy. Zazwyczaj deklaracja ma postać:

```
X&operator=(const X&)
```

- Operator ten nie jest dziedziczony, ponieważ musi skopiować *wszystkie* pola klasy.
- Dla wielu klas kompilator jest w stanie wygenerować automatyczny operator przypisania, który wywoła operatory przypisania poszczególnych atrybutów.

Przykład

```
class TwoStrings
{
public:
    String s1;
    String s2;
};

TwoStrings a,b;
a = b;
// automatycznie wywoła: //
// a.s1= b.s1;
// a.s2=b.s2;
```

Standardowa implementacja operatora przypisania dostarczona przez kompilator wywołuje operator przypisania dla kolejnych pól obiektu.

Jeżeli te pola nie są wskaźnikami, zazwyczaj implementacja własnego operatora przypisania nie jest konieczna.

Operator przypisania powinien być definiowany dla klas **alokujących pamięć**. Zazwyczaj także definiujemy wtedy konstruktor kopiujący.

Modelowy przykład klasy alokującej pamięć

```
class Array
{
    double *data;
    int size;
public:
    Array(int s=0):size(s),data(0){
        if(size>0)data = new double[size];
    }
    Array(const Array&other){
        copy(other);
    }
    ~Array(){free();}
    Array&operator=(const Array&other){
        if(&other != this){
            free();
            copy(other);
        }
        return *this;
    }
protected:
    void free();
    void copy(const Array&other);
};
```

Modelowy przykład klasy alokującej pamięć

```
void Array::free(){
    if(data)delete []data;
    data =0;
    size=0;
}
void Array::copy(const Array&other){
    size = other.size;data=0;
    if(size>0)data = new double[size];
    for(int i=0;i<size;i++)data[i]=other.data[i];
}
```

`if(&other != this)` – zabezpiecza przed zwolnieniem pamięci obiektu, dla bezpośredniego lub pośredniego wywołania przypisania `x = x`.

Operator wywołania funkcji

Tą nazwą określa się operator ( ) . Operator ten jest operatorem dwuargumentowym i ma postać

`expression (expression-list)`

`expression` – jest zazwyczaj nazwą funkcji,

`expression-list` – listą argumentów.

```
class Matrix
{
    double e[100][100];
public:
    double&operator()(int row,int col)
    {
        return e[row][col];
    }
};
```

```
int main(){
    Matrix m;
    m(2,3)=7.5;
    printf("%f",m(2,3));
}
```

Operatory ekstrakcji i wstawiania

Zazwyczaj definiuje się w celu odczytu lub zapisu zawartości obiektów z/do strumieni.

```
Vector v;
cin>>v;
cout<<v;
```

Ponieważ pierwszym argumentem jest strumień, operatory mogłyby być zaimplementowane:

- jako **metoda strumienia** z jednym parametrem typu `Vector&`
- jako **globalna funkcja z dwoma parametrami**

Ponieważ nie modyfikujemy klas bibliotecznych, **zawsze są implementowane jako funkcje globalne**.

Przykład

```
#include <iostream>
using namespace std;

class Array
{
    friend ostream&operator<<(ostream&os,const Array&v);
    friend istream&operator>>(istream&is, Array&v);
    double *data;
    int size;
public:
    // pozostałe deklaracje
};
```

Deklarujemy operatory jako funkcje zaprzyjaźnione (`friend`). W ten sposób będą miały dostęp do pól prywatnych.

Przykład

```
ostream&operator<<(ostream&os,const Array&v){
    os<<v.size<<" ";
    for(int i=0;i<v.size;i++){
        os<<v.data[i]<<" ";
    }
    return os;
}

istream&operator>>(istream&is, Array&v){
    if(!is)return is;
    v.free();
    is>>v.size;
    v.data = new double[v.size];
    for(int i=0;i<v.size;i++){
        is>>v.data[i];
    }
    return is;
}
```

Zapisujemy rozmiar,

a następnie wszystkie dane

1. Czyścimy zawartość,
2. Wczytujemy rozmiar
3. Alokujemy pamięć
4. Wczytujemy dane do tablicy