Język C++

część 7
przeciążanie operatorów

Jarosław Gramacki
Instytut Informatyki i Elektroniki

- przeciążanie operatora =
  - język C++ pozwala na zmianę zachowania się predefiniowanych operatorów języka w zależności od typów argumentów
  - do klasy dodamy więc funkcję `operator=()`, której zadaniem będzie kopiowanie zawartości pamięci wskazywanej przez `str` zamiast kopiowania tylko wskaźnika

```
class String {
public:
    String( const char* _str );
    String( int _size );
    ~String();
    void set( const char* _str );
    const char* get() const;
    void print() const;
    String& operator=( const String& _string ); // Operator przypisania.
private:
    int size;
    char* str;
};
String& String::operator=( const String& _string ) {
    if( this == &_string )                    // Sprawdza s1 = s1;
       return *this;

    delete [] str;                // Usuwa stary łańcuch.
    size = _string.size;
    str = new char[ size + 1 ];   // Przydziela niezbędna pamięć.
    strcpy( str, _string.str );   // Kopiuje zawartość łańcucha.

    return *this;
}
```

folia z wykładu #4

asekuracyjnie, aby obsłużyć wyrażenie takie jak np. `s1 = s1;`

lub: **this**.str

porównaj uwagę na kolejnym slajdzie

- przeciążanie operatora =
  - teraz obie instrukcje są równoważne

folia z wykładu #4

```
s2 = s1;
s2.operator=(s1);  // zwykłe wywołanie funkcji o nazwie operator=
```

operator = jest operatorem dwu-argumentowym. U nas, w wersji przeciążonej również. Pierwszy argument to obiekt na rzecz którego wywołujemy funkcję operator=(), drugi argument to argument tejże funkcji

  - łączenieć w jeden łańcuch operacji kilka kolejnych instrukcji przypisania, czyli korzystać z funkcjonalności analogicznej jak dla zwykłego operatora przypisania zastosowanego do zmiennych prostych typów

```
String s1( "String 1" );
String s2( "String 2" );
String s3( "String 3" );
s3 = s2 = s1;
```

referencje: porównaj też informacje podane na wykładzie 2

  - powyższe stwierdzenie łatwiej zrozumieć gdy użyjemy innej (ale równoważnej) formy (operator = jest łączny prawostronnie)

```
s3.operator=(s2.operator=(s1));
```

  - widać teraz wyraźnie, że argumentem funkcji `operator=()` jest wartość zwracana z poprzedniego wywołania funkcji `operator=()`. Oznacza to, że zwracany przez funkcję `operator=()` typ musi być akceptowany jako wejście tej samej funkcji
  - sygnatura domyślnej wersji funkcji `operator=()` wygląda następująco:

```
AClass& AClass::operator=(const AClass&);
```


- przeciążanie operatorów - wprowadzenie i podstawy
  - na jednym z poprzednich wykładów było omawiane przeciążanie operatora przypisania =
  - dotyczyło to sytuacji obsługi klasy `String`, która zawierała pole wskaźnikowe. Przeciążenie operatora było tam konieczne
  - operatory przeciąża się też dla wygody korzystania z obiektów danej klasy

"nasza" wersja

```
// tak jest prościej:

String imie(20);
cout << "Podaj imie: ";
cin >> imie;
if (imie == "Piotr") ...

String nazwisko(20);
cout << "Podaj nazwisko: ";
cin >> nazwisko;
if (nazwisko != "Kowalski"
    && nazwisko !="Nowak") ...

String osoba(40);
osoba = imie + " " + nazwisko;

cout << osoba << "[3]=" << osoba[3] << endl;
```

łatwy odczyt

Łatwe testowanie

Łatwe testowanie

Łatwe tworzenie nowego obiektu

Łatwe wyprowadzanie

Łatwy dostęp do 3-ego znaku

```
// zamiast:

String imie(20);
cout << "Podaj imie: ";
imie.Read();
if (imie.IsEqual("Piotr")) ...

String nazwisko(20);
cout << "Podaj nazwisko: ";
nazwisko.Read();
if (!nazwisko.IsEqual("Kowalski")
    && !nazwisko.IsEqual("Nowak")) ...

String osoba(40);
osoba.Copy(imie);
osoba.Concat(" ");
osoba.Concat(nazwisko);

osoba.Print();
cout << "[3]=" << osoba.GetChar(3) <<
endl;
```

Tu wszędzie odwołujemy się do funkcji zawartych w klasie `String`

- przeciążanie operatorów - wprowadzenie i podstawy
  - w wersji z przeciążaniem operatorów "rozszerzono" operatory:

    `>>` (pobieranie danych obiektu)

    `==` oraz `!=` (testowanie wartości)

    `=` oraz `+` ("sklejanie" obiektów)

    `<<` (wyświetlanie obiektu)

    `[]` (n-ty znak obiektu)
  - druga wersja programu jest bardzo "ociężała" mimo, że klasa `String` jest bardzo prosta. Dla "większych" klas problem będzie jeszcze bardziej widoczny. Przykład obiektowej obsługi liczb zespolonych:

```
// tak ?
Zespolona fun( Zespolona a, Zespolona b, Zespolona c)
{
// b*b - 4*a*c
return (sub(mult(b,b), mult(4, mult(a,c))); }

// ... czy tak ?
Zespolona fun( Zespolona a, Zespolona b, Zespolona c)
{
// b*b - 4*a*c
return (b*b - 4*a*c); }
```


- przeciążanie operatorów - obowiązujące zasady
  - nie można tworzyć nowych operatorów, można jedynie przeciążać (prawie wszystkie) operatory istniejące w języku C++
  - lista operatorów, których nie wolno przeciążać:
    `?:`
    `::`
    `.`
    `.*`
  - operatory w oryginale np. jednoargumentowe trzeba po przeciążeniu stosować również z jednym operandem np:
    `a = b ~ c` (błąd, operator `~` jest jednoargumentowy)
  - nie można zmieniać priorytetu i łączności (kierunku wiązania) operatorów. Można jednak wpływać na sposób obliczania wyrażenia stosując w odpowiednich miejscach nawiasy
  - nie można zmieniać zachowania się operatorów wobec typów predefiniowanych. Przykładowo nie można przeciążyć operatora `+` wobec zmiennej `int`. Przeciążanie może zachodzić tylko wobec nowych typów danych (klas)
  - przeciążanie nie jest domyślne w żadnym sensie. Przykładowo: przeciążenie operatora `+` nie przeciąża automatyczne operatora `+=`. To samo dla pary `==` i `!=`, itd.
  - najbardziej uzasadnione (praktyczne) jest przeciążanie operatorów klas matematycznych

- przeciążanie operatorów - przykład klasy String
  - Uwaga. Zajrzyj też do wykładu #4 "**konstruktory, destruktory**"

```
class String {
public:
    String( const char* _str );
    String( int _size );
    String( const String& _string );          // Konstruktor kopiujacy.
    ~String();
    void set( const char* _str );
    const char* get() const;
    void print() const;

    int operator==( const String& _string );     // if( s1 == s2 )
    String& operator=( const String& _string );  // s1 = s2;
    int operator!=( const String& _string );     // if( s1 != s2 )
    String operator+( const String& _string );   // s1 = s2 + s3;
    String& operator+=( const String& _string ); // s1 += s2;
    char& operator[]( int _indeks );             // c = s1[ 5 ];

    friend ostream& operator<<( ostream& _s, const String& _string );  // cout << s1;
    friend istream& operator>>( istream& _s, String& _string );        // cin >> s1;

private:
    int   size;
    char* str;
};
```

większość operatorów przeciążonych to funkcje składowe klasy `String`

porównaj z jakim argumentem następuje wywołanie tej metody na następnej stronie (z `char *` a nie z `const String&`)

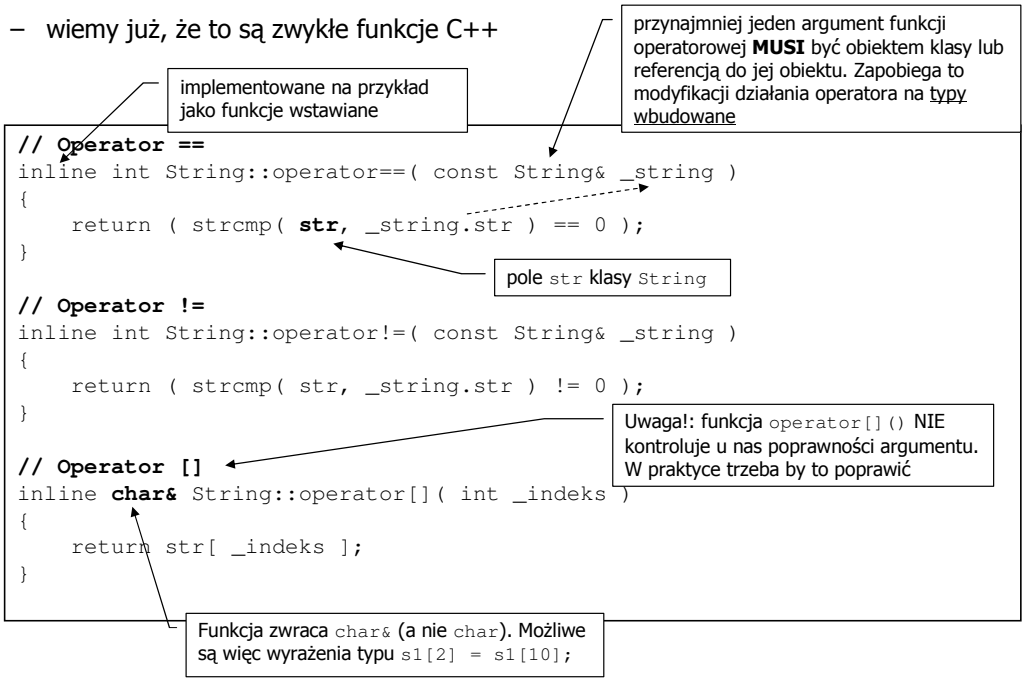
tu jest inaczej (to są funkcje **zaprzyjaźnione**). Wrócimy do tego w dalszej części wykładu

- przeciążanie operatorów - przykład klasy String
  - w komentarzach podano jak instrukcje programu zinterpretuje kompilator
  - kwestia użycia konstruktora jako funkcji umożliwiającej konwersję typów będzie omawiana później. Tu jest jedynie zasygnalizowana.

```
// tak:

String imie(20);
cout << "Podaj imie: ";
cin >> imie;                          // operator>> (cin, imie);
if (imie == "Piotr") ...              // if (imie.operator== ("Piotr")) ...

String nazwisko(20);
cout << "Podaj nazwisko: ";
cin >> nazwisko;
if (nazwisko != "Kowalski"            // if (nazwisko.operator!= ("Kowalski")
    && nazwisko !="Nowak") ...

String osoba(40);
osoba = imie + " " + nazwisko;        // osoba.operator= ((imie.operator+
                                      // (" ")).operator+ (nazwisko));

cout << osoba << "[3]=" << osoba[3] << endl;
// operator<< (cout, osoba);
// cout << "[3]=" << osoba.operator[](3) << endl;
```

Uwaga! Tu w rzeczywistości odbywa się konwersja z typu `char*` na typ `String`
Następuje użycie konstruktora celem utworzenia chwilowego obiektu typu String:
`if (imie.operator== ( String("Piotr") ) ...`

- przeciążanie operatorów – implementacja
  - implementacja funkcji **`operator==(), operator!=(), operator[]()`**
  - wiemy już, że to są zwykłe funkcje C++




```
// Operator ==
inline int String::operator==( const String& _string )
{
    return ( strcmp( str, _string.str ) == 0 );
}

// Operator !=
inline int String::operator!=( const String& _string )
{
    return ( strcmp( str, _string.str ) != 0 );
}

// Operator []
inline char& String::operator[]( int _indeks )
{
    return str[ _indeks ];
}
```


- przeciążanie operatorów - implementacja
  - implementacja funkcji **`operator+=()`**
  - dzięki tej funkcji można będzie użyć np. takich wywołań

```
String s1("Jeden-"), s2("Dwa");
s1 += s2; // s1 = "Jeden-Dwa"
```

```
// Operator +=
String& String::operator+=( const String& _string )
{
    size += _string.size;

    // Przydzielenie potrzebnej pamieci.
    char *ps = new char[ size + 1 ];

    // Utworzenie lancucha wynikowego poprzez konkatenacje
    // lancuchow zrodlowych.
    strcpy( ps, str );
    strcat( ps, _string.str );

    // Zwolnienie bufora poprzedniego String i
    // zastapienie nowym obszarem pamieci.
    delete [] str;
    str = ps;

    return *this;
}
```

- przeciążanie operatorów – implementacja
  - implementacja funkcji **`operator+()`**
  - używamy w niej wcześniej napisanej funkcji `operator+=()` gdyż używamy odpowiednio operatora `+=`
  - funkcja `operator+()` NIE modyfikuje obiektu, w stosunku do którego została użyta !
  - funkcja `operator+()` tworzy i zwraca obiekt typu `String` (nie `String&`) jako złożenie argumentu pobranego z argumentem, wobec którego jej użyto

```
s1 + s2; // s1 jest obiektem, wobec którego użyto funkcji operator+()
         // s2 jest argumentem tej funkcji
```

Uwaga! W poprzednich przykładach tu było **String&**

Funkcja tworzy obiekt chwilowy i zwraca sam obiekt, a nie referencję do niego

```
// Operator +
String String::operator+( const String& _string )
{
    String temp = *this;     // Kopiuje bieżący obiekt String.
    temp += _string;         // Konkatenuje drugi obiekt String.
    return temp;             // Zwraca utworzony łańcuch.
}
```


- przeciążanie operatorów – operatory << i >>
  - Operatory << i >> nie mogą być funkcjami składowymi klasy `String`, gdyż nie stosują się do obiektów `String` ale do obiektów typu `ostream` i `istream` (porównaj: **`cin`** `>> imie`)
  - konieczne jest więc definiowanie tych operatorów jako funkcji zaprzyjaźnionych klasy `String`
  - funkcje te będą miały więc dostęp do komponentów `private` obiektów typu `String` (a o to nam chodzi)
  - Każda funkcja zwraca referencję do strumienia, który pobrała jako argument i który z kolei także został przekazany przez referencję. Możliwe są więc wyrażenia typu: `cout << "Hello, " << nazwisko << " !" << endl;`

```
// Operator <<
ostream& operator<<( ostream& _s, const String& _string )
{
    return ( _s << _string.str );       // "opakowanie" jednej czynności w
}                                       // w funkcję operatorową

// Operator >>
istream& operator>>( istream& _s, String& _string )
{
    String temp( 100 );                 // Bufor wejścia.

    _s >> temp.str;
    _string = temp;                     // Wykorzystujemy operator=

    return _s;
}
```

Uwaga! brak prefiksu `String` `(String::operator<< ...)`

- przeciążanie operatorów – operatory << i >>

```
// interpretacja kompilatora
operator<<(
   operator<<(
      operator<<(
         operator<<( cout, "Hello, "),
                 ),
      "!"
      ),
    endl
    );
```

Tak kompilator zinterpretuje wyrażenie:
`cout << "Hello, " << nazwisko << " !" << endl;`

- przeciążanie operatorów – funkcje zaprzyjaźnione raz jeszcze
  - czasami trzeba rozważyć dodatkowe kwestie użycie funkcji zaprzyjaźnionych w implementacji funkcji przeciążających operatory
  - pokazane poniżej problemy wynikają z faktu, że funkcji składowej pewnej klasy NIE można używać w stosunku do obiektu, który nie jest obiektem tejże klasy (`"Napis"` NIE jest obiektem typu `String`)

```
String s1, s2;

if (s1 == s2);              // OK. Interpretacja: s1.operator==(s2)
if (s1 == "Napis");         // OK. j.w.

if ("Napis" == s1);         // Błąd. Jak zinterpretować: "Napis".operator==(s1)
String s2 = "Napis " + s1; // Błąd. Jak zinterpretować: "Napis".operator+(s1)
```

tylko pozornie niegroźna zmiana kolejności

  - używając konstruktora `String(char*)` możemy konwertować obiekty typu `char` w `String`
  - zmodyfikujmy więc funkcję `operator==()` do postaci:
    `int operator== (const String& _s1, const& String _s2);`
    oraz uczyńmy ją zaprzyjaźnioną z klasą `String` i wtedy:

teraz dwa arumenty

```
String s1, s2;

if (s1 == s2);              // if (operator==( s1, s2 ))
if (s1 == "Napis");         // if (operator==( s1, String("Napis") ))
if ("Napis" == s1);         // if (operator==( String("Napis"), s2 ))
```

- przeciążanie operatorów – funkcje zaprzyjaźnione raz jeszcze
  - ostateczna wersja funkcji **`operator==()`** oraz (przez analogię) funkcji **`operator!=()`** i **`operator+()`**

```
class String  {
public:
    // ...
    friend int operator==( const String& _s1, const String& _s2 );
    friend int operator!=( const String& _s1, const String& _s2 );
    friend String operator+( const String& _s1, const String& _s2 );
    String& operator+=( const String& _string );
    char& operator[]( int _indeks );
    friend ostream& operator<<( ostream& _s, const String& _string );
    friend istream& operator>>( istream& _s, String& _string );
    // ...
};

// Operator ==
inline int operator==( const String& _s1, const String& _s2 ) {
    return ( strcmp( _s1.str, _s2.str ) == 0 );
}
// Operator !=
inline int operator!=( const String& _s1, const String& _s2 ) {
    return ( strcmp( _s1.str, _s2.str ) != 0 );
}
// Operator +
String operator+( const String& _s1, const String& _s2 ) {
    String temp = _s1;      // Kopiuje argument z lewej strony.
    temp += _s2;            // Dolacza argument z prawej strony
    return temp;            // Zwraca utworzony lancuch. }
```

Tu funkcje operatorowe nie są elementem klasy. Dlatego posiadają teraz DWA argumenty (nie można używać wskaźnika `this`)

Jak więc widać, wybór czy stosować funkcję składową klasy czy funkcje zaprzyjaźnioną jest skomplikowanym zagadnieniem !!!

- przeciążanie operatorów – operatory jednoargumentowe
  - o. jednoargumentowy można przeciążać za pomocą funkcji składowej klasy nie pobierającej żadnych argumentów lub
  - o. jednoargumentowy można przeciążać za pomocą funkcji friend pobierającej jeden argument (obiekt lub referencja do obiektu)
  - przykład przeciążenia operatora **`!`**
  - chcemy sprawdzić, czy obiekt klasy `String` jest pusty

Dla operatorów dwuargumentowych tak nie było; sprawdź sam

```
// 1. sposób
class String {
public:
  bool operator!() const;              // czy napis jest pusty
  ...
};

bool String::operator!() const { return size == 0; }

String s("Jarek");
if( !s )             // s.operator!()
```

Patrz definicja klasy `String`

```
// 2. sposób
class String {
public:
  friend bool operator!(const String&);  // czy napis jest pusty
  ...
};

String s("Jarek");
if( !s )             // operator!(s)
```

Uwaga! brak prefiksu `String` (`String::operator! ...`)

- przeciążanie operatorów – operatory jednoargumentowe
  - problemy z operatorami `++` oraz `--`
  - operatory te mogą być w wersji prefiksowej lub postfiksowej i tu leży problem. Jak kompilator ma je rozróżnić gdy zostaną przeciążone ?
  - funkcja operatorowa musi posiadać więc jakąś sygnaturę, po której kompilator odróżni, która wersja operatora ma być użyta
  - zasada jest następująca:
    wersja prefiksowa przeciążana jest tak samo jak inne prefiksowe operatory 1-argumentowe
    wersja postfiksowa jest identyfikowana dzięki dodatkowemu argumentowi, który musi być typu `int`.

- przeciążanie operatorów – operatory jednoargumentowe

```
class Date {
...
public:
   Date( int m = 1, int d = 1, int y = 1900 ); // constructor
   Date &operator++();            // preincrement operator
   Date operator++( int );        // postincrement operator
   ...
private:
   int year;
   int month;
   int day;
   void helpIncrement();          // utility function
};

void Date::helpIncrement() {
  ++year; ++month; ++day; }

Date& Date::operator++() {
   helpIncrement();
   return *this;  // reference return to create an lvalue
}

// overloaded postincrement operator; note that the dummy
// integer parameter does not have a parameter name
Date Date::operator++( int ) {
   Date temp = *this;
   helpIncrement();

   // return unincremented, saved, temporary object
   return temp;   // value return; not a reference return
}
```

Omówienie tej funkcji na następnej stronie

- przeciążanie operatorów – operatory jednoargumentowe
  - opis funkcji **`Date operator++( int );`**
  - aby zasymulować efekt zwiększenia daty PO wykonaniu operacji, powinniśmy zwrócić taką kopię obiektu, której zawartość NIE została jeszcze powiększona
  - na początku zapamiętujemy więc aktualny obiekt (`*this`) w zmiennej pomocniczej (`temp`).
  - następnie wywołujemy funkcję pomocniczą `helpIncrement()`, która inkrementuje datę i na koniec zwracamy niepowiększoną kopię obiektu
  - uwaga: nie można w tej funkcji zwrócić referencji gdyż zmienne lokalne są usuwane, gdy funkcja, w której zostały zadeklarowane, kończy swoje działanie
  - (zwracanie referencji do zmiennej lokalnej to typowy błąd !)

- przeciążanie operatorów – operatory jednoargumentowe

```
int main()
{
   Date d4( 12, 27, 1992 );
   cout << "\n\nTesting the preincrement operator:\n"
        << "  d4 is " << d4 << '\n';
   cout << "++d4 is " << ++d4 << '\n';   // d4.operator++()
   cout << "  d4 is " << d4;

   cout << "\n\nTesting the postincrement operator:\n"
        << "  d4 is " << d4 << '\n';
   cout << "d4++ is " << d4++ << '\n';  // d4.operator++( 0 )
   cout << "  d4 is " << d4 << endl;

   return 0
}
```

lub `operator++(d4)` gdy funkcja `friend`

lub `operator++(d4, 0)` gdy funkcja `friend`

- przeciążanie operatorów – przykład kończący

```
#include <iostream.h>

class Matrix
{
  private:
    int rows, cols;
    int **p;    // base of array
  public:
    Matrix(void);
    Matrix(int, int);
    ~Matrix(void);
    void LoadMat(void);
    void DisplayMat(void);
    void   operator= (Matrix&);
    Matrix operator+ (Matrix&);
    Matrix operator- (Matrix&);
};
```

```
// lub bardziej rzeczywisty przykład
...
Matrix operator - ();
Matrix operator - (double value);
Matrix operator - (Matrix& matrix);
Matrix operator -= (double value);
Matrix operator -= (Matrix& matrix);
Matrix& operator = (Matrix& amatrix);
Matrix operator + (double value);
Matrix operator + (Matrix& matrix);
Matrix operator += (double value);
Matrix operator += (Matrix& matrix);
Matrix operator * (Matrix& matrix);
Matrix operator *= (Matrix& matrix);
Matrix operator ^ (double value);
Matrix operator ^= (double value);
bool operator == (Matrix& matrix);
bool operator != (Matrix& matrix);
friend istream& operator >> (istream& s, Matrix& matrix);
friend ostream& operator << (ostream& s, Matrix& matrix);
double& operator[](int pos);
...
```


- przeciążanie operatorów – przykład kończący

```
Matrix::Matrix(void) {
    cout << "Enter the number of rows";
    cin >> rows;
    cout << "Enter the number of columns";
    cin >> cols;
    p = new *[rows];
    for (int i=0; i < rows; ++i)
        p[i] = new int[cols];
}
Matrix::Matrix(int inrows, int incols) {
    rows = inrows;
    cols = incols;
    p = new *[rows];
    for (int i=0; i < rows; ++i)
        p[i] = new int[cols];
    cout << " Contructor " << p << "\n";
}

Matrix::~Matrix(void) {
    cout << " Destructor " << p << "\n";
    for (int i=0; i < rows; ++i)
        delete [] p[i];
    delete [] p;
}
```

- przeciążanie operatorów – przykład kończący

```
void Matrix::DisplayMat(void) {
    for (int i=0; i < rows; ++i) {
        for (int j=0; j < cols; ++j) {
            cout << p[i][j] << " ";
        }
        cout << "\n";
    }
    cout << "\n";
}

void Matrix::LoadMat(void) {
    for (int i=0; i < rows; ++i)
        p[i] = new int[cols];
    for (int j=0; j<rows; ++j) {
        for (int k=0; k < cols; ++k) {
            cout << "Enter (" << j << "," << k << ")";
            cin >> p[j][k];
        }
    }
}
```


- przeciążanie operatorów – przykład kończący

```
void Matrix::operator= (Matrix& RightOp) {
    if (rows == RightOp.rows && cols == RightOp.cols) {
        for (int i=0; i < rows; ++i) {
            for (int j=0; j < cols; ++j) {
                p[i][j] = RightOp.p[i][j];
            }
        }
    }
    else {
        delete p;
        rows = RightOp.rows;
        cols = RightOp.cols;
        p = new *[rows];
        for (int i=0; i < rows; ++i)
            p[i] = new int[cols];
        for (int j=0; j<rows; ++j) {
            for (int k=0; k < cols; ++k) {
                p[j][k] = RightOp.p[j][k];
            }
        }
    }
}
```

- przeciążanie operatorów – przykład kończący

```
Matrix Matrix::operator+ (Matrix& RightOp) {
    if (rows == RightOp.rows && cols == RightOp.cols) {
        Matrix temp(rows,cols);
        for (int i=0; i < rows; ++i) {
            for (int j=0; j < cols; ++j) {
                temp.p[i][j] = p[i][j] + RightOp.p[i][j]; }
        }
        return (temp);
    }
    else
        cout << "Mismatched Matrix sizes in addition function\n";
    }

Matrix Matrix::operator- (Matrix& RightOp) {
    if (rows == RightOp.rows && cols == RightOp.cols) {
        Matrix temp(rows,cols);
        for (int i=0; i < rows; ++i) {
            for (int j=0; j < cols; ++j) {
                temp.p[i][j] = p[i][j] - RightOp.p[i][j]; }
        }
        return (temp);
    }
    else
        cout << "Mismatched Matrix sizes in subtraction function\n";
    }
```


- przeciążanie operatorów – przykład kończący

```
// a wszystko dla tej wygody ...

main() {

    Matrix A(2,2), B(2,2), C(2,2);

    A.LoadMat();
    B.LoadMat();

    C = A + B - A + C;

    A.DisplayMat();
    B.DisplayMat();
    C.DisplayMat();

}
```

Przeciążanie operatorów uważane jest za trudne zagadnienie. W języku **Java** w ogóle z tego zrezygnowano.