Programowanie obiektowe

Paweł Mikołajczak, 2019

5. Przeciążanie operatorów

Web site: informatyka.umcs.lublin.pl

Przeciążanie operatorów

- ***Wstęp***
- ***Definicje***
- ***Przeciążony operator dodawania (+).***
- ***Przeciążony operator mnożenia (*).***
- ***Funkcja operatorowa w postaci niezależnej funkcji***
- ***Przeciążanie operatorów równości i nierówności.***
- ***Przeciążanie operatora przypisania (=)***
- ***Przeciążanie operatora wstawiania do strumienia (<<)***

Przeciążanie operatorów

Programując w języku C++ programista może korzystać zarówno z typów wbudowanych jak i typów stworzonych przez siebie.

W języku C++ został ***zdefiniowany zestaw operatorów,*** wszystkie one są używane w połączeniu z typami wbudowanymi. W języku C++ nie można tworzyć nowych operatorów. Często jednak zachodzi potrzeba ***zmienienia sposobu działania*** konkretnego operatora.

Język C++ pozwala na ***przeciążanie istniejących operatorów***, to znaczy możemy nadawać im nowe znaczenie dla operacji wykonywanych częściowo lub w całości na obiektach typu klasa. Być może przeciążanie operatorów na pierwszy rzut oka jest dość egzotyczną techniką, ale wielu programistów nie zdaje sobie sprawy z tego, że używa przeciążonych operatorów. Przykładem przeciążonego operatora jest operator dodawania (**+**). Operator dodawania działa inaczej na danych typu **int** i inaczej na danych typu **double**. Takie działanie jest możliwe, ponieważ operator dodawania został przeciążony w samym języku C++.

Przeciążanie operatorów

Aby przeciążyć operator definiuje się funkcję (z nagłówkiem i ciałem) w zwykłej postaci z wyjątkiem tego, że nazwą funkcji jest słowo kluczowe operator poprzedzające symbol przeciążanego operatora.

Na przykład nazwa funkcji **operator+** mogłaby być użyta do przeciążenie operatora dodawania (**+**). Przeciążanie operatora realizowane jest:

- albo w postaci niezależnej funkcji (zazwyczaj zaprzyjaźnionej z jedną lub kilkoma klasami)
- albo w postaci funkcji składowej

W pierwszym przypadku, jeżeli **oper** jest operatorem dwuargumentowym, to zapis:

x oper y

jest równoznaczna wywołaniu:

operator oper (x, y)

w drugim przypadku ten sam zapis odpowiada wywołaniu:

x.operator oper (y)

Przeciążanie operatorów

Przeciążony operator musi mieć
jako jeden z operandów obiekt.

W języku C++ można przeciążać prawie wszystkie operatory, a lista operatorów, które mogą być przeciążane jest prawie kompletna. Każdy z tych operatorów może być przeciążony w dowolny sposób (np. operator dodawania po przeciążeniu wcale nie musi wykonywać operacji dodawania). Oczywiście są pewne ścisłe reguły. Operator binarny musi pozostać binarnym, a operator unarny musi pozostać unarnym.

Działanie symboli operatorów może być przedefiniowane tak, aby jak najlepiej obsługiwać swoją klasę, z tym, że należy pamiętać o wielu ograniczeniach.

Przeciążony operator

Niektóre operatory posiadają specjalne właściwości, do takich należy operator przypisania (=).

Domyślny operator przypisania zdefiniowany jest dla każdej klasy. Przypisuje on poszczególne składowe obiektów i zwraca obiekt, któremu przypisana została nowa wartość. Przeciążanie operatora przypisania wymaga rozwiązania kilku ważnych kwestii, szczególnie, gdy składowymi klas są wskaźniki. Istnieją także inne operatory ***automatycznie generowane*** dla każdej klasy.

Są to:

operator pobrania adresu (**&**), obiektu danej klasy

operator przecinka (**,**)

operator tworzenia obiektów dynamicznych (**new**)

operator niszczenia obiektów dynamicznych (**delete**)

Przeciążanie operatorów

Symbole, które nie mogą być przeciążane są pokazane poniżej:

- operator kropka(.)
- operator wyłuskania wskaźnika do składowej (.*)
- operator zasięgu (::)
- operator warunkowy (?:)
- operator rozmiaru (sizeof)
- symbole # i ##

Nie można kreować nowych symboli operatorów.

Np. ^^ nie jest operatorem w C++, nie może być kreowany jako operator klasy.

W języku C++ nie istnieje operator potęgowania, więc próba tworzenia własnego operatora potęgowania nie wydaje się konieczna, w języku istnieje odpowiednia funkcja, którą należy wywołać w celu wykonania potęgowania.

Przeciążanie operatorów

- Ani kolejność (ang. ***precedence***) ani łączność (ang. ***associativity***) operatorów C++ nie może być modyfikowana. Nie można np. operatorowi dodawania nadać wyższy priorytet niż ma operator dzielenia.

- Nie można redefiniować operatorów dla typów wbudowanych

- Operator unarny nie może być zmieniany na binarny, a binarny nie może być zmieniany na unarny.

- Operator musi być albo członkiem klasy albo być tak definiowany, aby przynajmniej jeden członek klasy był jego operandem.

Przeciążanie operatorów

Zazwyczaj konstruując jakąś klasę należy określić, czy potrzebne będą specjalne operatory. Na przykład, aby porównać dwa trójwymiarowe wektory, należy kolejno sprawdzić czy odpowiednie składowe (x, y i z) dwóch wektorów są równe. Zdefiniowana przez użytkownika ***operacja*** (operator) jest projektowana jako funkcja, która redefiniuje wbudowany w C++ symbol operatora do wykorzystania przez klasę.

Funkcja, która definiuje operacje na obiektach klasy i wykorzystuje wbudowane w C++ symbole operatorów nosi nazwę **funkcji operatorowej** (ang. ***operator function***).

Funkcje operatorowe są deklarowane i implementowane w taki sam sposób, jak wszystkie funkcje składowe, z jednym wyjątkiem – wszystkie funkcje operatorowe mają nazwę postaci:

operator op

gdzie **op** jest jednym z symboli pokazanych w tabeli przeciążanych operatorów.

Na przykład, nazwa funkcji **operator +** jest nazwą funkcji dodawania, a nazwa funkcji **operator ==** jest nazwą funkcji porównującej „**równy z**”.

Przeciążony operator dodawania (+).

Omówimy prosty przykład ilustrujący działanie przeciążonego operatora **dodawania.** Funkcja operatora jest funkcją składową klasy.

Przykładowy program używa klasy **wek** i funkcji składowej, która jest przeciążonym operatorem dodawania. W omawianym programie wykonywane jest dodawanie dwóch wektorów (dwu-wymiarowych), w wyniku otrzymuje trzeci wektor. Operacja dodawania wektorów polega na oddzielnym dodawaniu składowych poszczególnych wektorów:

$$\vec{a} + \vec{b} = \vec{c}$$

$$a_x + b_x = c_x$$

$$a_y + b_y = c_y$$

Przeciążony operator dodawania (+).

Zadanie dodawania wektorów zrealizujemy za pomocą operatora +, przeciążonego w taki sposób, aby wykonać pokazaną metodą dodawanie składowych wektorów. W konwencji używanej przez ten program po dodaniu wektora **a** do wektora **b** otrzymamy wektor **c.**

Opracowana klasa ma postać:

```
class wek
{     int vx, vy;
  public:
      void ustaw( int ux, int uy);
      void pokaz();
      wek operator+ (wek v);
};
```

Przeciążony operator dodawania (+).

W deklaracji klasy **wek** mamy także funkcję operatorową (funkcja przeciążonego operatora), która nadaje nowe znaczenie operatowi +:

wek operator+ (wek v);

Definicja funkcji operatorowej ma postać:

```
wek wek :: operator+ ( wek v )
    { wek wektor;
      wektor.vx = vx + v.vx;
      wektor.vy = vy + v.vy;
      return wektor;
    }
```

Należy zauważyć, że w tej deklaracji pierwszym argumentem operatora jest wektor, (ponieważ operator jest składową klasy **wek**), a drugim argumentem jest także wektor (ponieważ typem parametru jest **wek**). **wek** przed słowem kluczowym ***operator*** oznacza, że rezultatem użycia operatora (zwracanym przez niego typem) jest nowy wektor.

Przeciążony operator dodawania (+)

```
#include <iostream>
#include <conio>
using namespace std;
class wek
{ int vx, vy;
  public:
   void ustaw( int ux, int uy);
   void pokaz();
   wek operator+ (wek v);
};
//--------------------------------
void wek :: ustaw(int ux, int uy)
{ vx = ux;    vy = uy;
}
void wek :: pokaz()
{ cout << "skladowe wektora: ";
  cout << vx << ", " << vy << endl;
}
wek wek :: operator+ ( wek v )
{ wek wektor;
  wektor.vx = vx + v.vx;
  wektor.vy = vy + v.vy;
  return wektor;
}
```

```
int main()
{
 wek a, b, suma_wek;
 a.ustaw(1, 1);
 b.ustaw(1, 2);
 a.pokaz();
 b.pokaz();
 cout << "po dodaniu ";
 suma_wek = a + b;
      // suma_wek = a.operator+(b);
 suma_wek.pokaz();
  getche();
  return 0;
}
```

Wydruk z programu ma postać:

skladowe wektora: 1, 1
skladowe wektora: 1, 2
po dodaniu skladowe wektora: 2, 3

Przeciążony operator dodawania (+).

W funkcji **main()** definiujemy trzy egzemplarze klasy ***wek***, **a**, **b** i **suma_wek**. Dane składowe wektorów **a** i **b** inicjalizujemy za pomocą funkcji **ustaw().** Do zmiennej **suma_wek** przypisujemy wynik wywołania przeciążonego operatora:

suma_wek = a + b;

Należy zauważyć, że funkcja operatorowa ma tylko jeden parametr, a wiemy, że przeciążamy operator dwuargumentowy. W języku C++ obowiązuje zasada, że jeżeli operator dwuargumentowy jest przeciążany za pomocą funkcji składowej klasy, to ***jawnie*** przekazywany jest mu tylko jeden argument. Drugi argument jest przekazywany ***niejawnie*** za pomocą wskaźnika **this.** Tak więc, zmienna **vx** występująca w instrukcji funkcji **operator+()**:

wektor.vx = vx + v.vx;

oznacza **this -> vx,** czyli element **vx** obiektu, który spowodował wywołanie funkcji operatora.

Przeciążony operator dodawania (+).

Należy zapamiętać następującą regułę:

Obiekt powodujący wywołanie funkcji operatora znajduje się zawsze po ***lewej stronie*** operacji. Obiekt występujący po ***prawej stronie*** jest przekazywany funkcji w postaci argumentu.

Jeżeli funkcja operatorowa jest składową klasy to w przypadku przeciążenia operatora jednoargumentowego nie ma potrzeby przesyłania argumentów.

W obu sytuacjach (przeciążanie operatorów jednoargumentowych i przeciążanie operatorów dwuargumentowych) obiekt powodujący wywołanie funkcji operatora jest jej niejawnie przekazywany za pomocą wskaźnika **this**. W definicji funkcji operatorowej w instrukcji:

wektor.vx = vx + v.vx;

vx jest prywatną daną klasy **wek**, udostępnioną za pomocą niejawnego wskaźnika **this**. Tą instrukcję możemy zapisać z jawnym wskaźnikiem **this**:

wektor.vx = this->vx + v.vx;

Przeciążony operator dodawania (+).

Z kolei w funkcji testującej instrukcję:

```
suma_wek = a + b;
```

możemy zastąpić ekwiwalentną instrukcją:

```
suma_wek = a.operator+(b);
```

Ten ostatni zapis pomaga zrozumieć, w jaki sposób funkcja **operator+()** otrzymuje niejawny wskaźnik **this**, odnoszący się do obiektu klasy **a**.

Za pomocą instrukcji

```
suma_wek = a + b;
```

wektor **suma_wek** otrzymuje wartości składowych, które są suma poszczególnych składowych wektorów **a** i **b**. Wykonanie pokazanej instrukcji składa się z dwóch operacji:

1. Najpierw wywołany jest operator dodawania (**+**), którego dwoma argumentami są wektory **a** i **b**. Operator dodawania zwraca nowy tymczasowy obiekt wektora jako wynik swojego działania.
2. Nowy wektor zwrócony przez operator dodawania jest przypisany wektorowi **suma_wek** za pomocą domyślnego operatora przypisania, który kopiuje składowe wektorów.

Przeciążony operator dodawania (+)

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

class punkt
{  int x,y;    //prywatne
  public:
    punkt() {x=0; y=0;}  //konstruktory
    punkt(int xx, int yy) {x = xx; y=yy;}
    int getx(){return x;} //akcesory
    int gety(){return y;}
    punkt operator+ (punkt); //funkcja operatorowa
};

punkt punkt::operator +(punkt p)
{   return punkt(x+p.x, y+p.y);
}

int main()
{   punkt p1(1,2),p2(3,4),p3;
    p3 = p1 + p2;
    cout << "p3, x= "<< p3.getx()<<endl;
    cout << "p3, y= "<< p3.gety()<<endl;
    return 0;
}
```

Wynik:
p3, x= 4
p3, y= 6

Przeciążony operator dodawania (+)

W programie mamy klasę punkt :

```
class punkt
{  int x,y;    //prywatne
  public:
    punkt() {x=0; y=0;}       //konstruktory
    punkt(int xx, int yy) {x = xx; y=yy;}
    int getx(){return x;}       //akcesory
    int gety(){return y;}
    punkt operator+ (punkt);       //funkcja operatorowa
};
```

W której znajduje się definicja funkcji operatorowej:

```
punkt punkt::operator +(punkt p)
{   return punkt(x+p.x, y+p.y);
}
```

W funkcji main() tworzymy trzy obiekty klasy punkt:

```
punkt p1(1,2),p2(3,4),p3;
```

W instrukcji:

```
p3 = p1 + p2;
```

wykorzystujemy przeciążony operator +

Przeciążony operator dodawania (+)

Tworzone klasy w bibliotece standardowej są bardzo rozbudowane, większość z nich posiada kolekcje przeciążanych operatorów.

Bardzo użyteczna jest klas string (plik nagłówkowy <string>). Między innymi mamy tam przeciążone operatory dodawania (+) oraz relacyjne , np. większości (>).

```
#include <iostream>
#include <string>

using namespace std;

int main()
{ string s1, s2;
  cout << "podaj imie ";
  cin >> s1;
  cout << "podaj nazwisko ";
  cin >> s2;
  cout << "lokator : "<< s1 + " " + s2 << endl;

  return 0;
}
```

Wynik:
podaj imie Jan
podaj nazwisko Kowalski
lokator : Jan Kowalski

Przeciążony operator relacyjny

Przeciążony operator porównania.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{ string s1, s2;
  cout << "podaj nazwisko ";
  cin >> s1;
  cout << "podaj nazwisko ";
  cin >> s2;
  if (s1 > s2)
     cout << s2 << " " << s1 << endl;
  else
     cout << s1 << " " << s2 << endl;

  return 0;
}
```

Wynik:
podaj nazwisko Kowalski
podaj nazwisko Arabski
Arabski Kowalski

Uwaga! Należy obsłużyć błąd!

Wynik:
podaj nazwisko arabski
podaj nazwisko Kowalski
Kowalski arabski

Przeciążony operator mnożenia (*)

Przeciążanie operatora mnożenia (*) zilustrujemy przykładem, w którym funkcja operatorowa (**operator* ( )**) umożliwi mnożenie dwóch ułamków.

Funkcja operatorowa jest ***funkcją składową*** klasy.

W pokazanym przykładzie klasa **ulamek** ma postać:

```
class ulamek
{     int li, mia;
  public:
      ulamek();
      ulamek (int, int);
      ulamek operator * (ulamek);
      void pokaz();
};
```

Ułamek przedstawiamy w postaci licznik/mianownik, wobec czego klasa **ulamek** ma dwie dane prywatne **li** i **mia** (licznik i mianownik). Klasa ma dwa konstruktory oraz funkcję składową **pokaz()** do wyświetlania licznika i mianownika ułamka.

Przeciążony operator mnożenia (*)

Deklaracje konstruktorów są następujące:

```
ulamek();
ulamek (int, int);
```

Domyślny konstruktor **ulamek()** ma postać:

```
ulamek :: ulamek(): li(0), mia(1)        //inicjuje ulamek zerem
{ }
```

i inicjuje ułamek 0/1 (licznik jest równy zeru, mianownik jest równy jeden).

W definicji konstruktora wykorzystaliśmy konstrukcję z tak zwaną ***listą inicjalizacji***.

Po nazwie konstruktora umieszczamy znak dwukropka, po nim pojawia się lista inicjalizowanych atrybutów. Powyższa definicja konstruktora równoważna jest klasycznej konstrukcji:

```
ulamek :: ulamek()
{
  li = 0;
  mia = 1;
}
```

Przeciążony operator mnożenia (*)

Zauważmy, że w języku C++ możemy deklarować zmienne i inicjować je na trzy sposoby:

a) int x = 0;
b) int x(0);
c) int x;
   x = 0;

Drugi konstruktor posiada dwa parametry i wywoływany jest wtedy, gdy tworzymy konkretny ułamek podając wartość licznika i mianownika. Konstruktor sprawdza także, czy mianownik nie jest zerem:

```
ulamek :: ulamek ( int a, int b)
{ if (b == 0)
    { cerr << "mianownik = 0 " << endl;
      exit(EXIT_FAILURE);
    }
      li = a;    mia = b;
}
```

Przeciążony operator mnożenia (*)

W klasie **ulamek** mamy także deklarowaną funkcję operatorową:

```
ulamek operator * (ulamek);
```

Widzimy, że pierwszym argumentem operatora mnożenia jest ułamek, ponieważ deklaracja funkcji operatorowej jest wewnątrz klasy **ulamek**. Drugim argumentem operatora jest także ułamek, ponieważ ten typ został umieszczony na liście argumentów operatora. Rezultatem zastosowania operatora mnożenia będzie nowy ułamek, ponieważ typ **ulamek** poprzedza słowo kluczowe **operator**.

Implementacja operatora mnożenia ma postać:

```
ulamek ulamek :: operator * ( ulamek x )
 {
     return ulamek ( li * x.li, mia * x.mia ) ;
}
```

Operator mnożenia zdefiniowany w klasie **ulamek** jest operatorem dwuargumentowym. Zastosowanie operatora zakresu postaci **ulamek ::** sprawia, że pierwszym argumentem operatora mnożenia jest obiekt klasy **ulamek**, na rzecz którego operator został wywołany. Drugi argument jest umieszczony na liście argumentów operatora mnożenia – jest to także obiekt klasy **ulamek**.

Przeciążony operator mnożenia (*)

W programie testującym tworzone są dwa ułamki, korzystamy z konstruktorów:

```
ulamek u1(3, 4), u2(5, 6);
```

Tworzone są dwa obiekty: **u1** (ułamek 3/4) oraz **u2** (ułamek 5/6).

Deklarujemy jeszcze jeden ułamek i wykonujemy mnożenie ułamków:

```
ulamek wynik ;
wynik = u1 * u2;
```

W programowaniu obiektowym nie istnieje globalna funkcja mnożenia dwóch ułamków, możemy wykonać jedynie odpowiednią operację. Do istniejącego ułamka wysyłany jest komunikat: „wykonaj mnożenie z przekazanym za pomocą parametru ułamkiem”. W naszym przypadku komunikatem tym jest funkcja operatorowa, a odbiorcą tego komunikatu jest ułamek.

Wyrażenie postaci:

```
u1 * u2
```

interpretowane jest jak:

```
u1.operator*(u2)
```

Przeciążony operator mnożenia (*)

W kontekście naszego programu operator mnożenia (*) mnoży dwa ułamki, jest to operator dwuargumentowy. Pierwszym argumentem jest ułamek **u1** (jest to obiekt, na rzecz którego została wywołana funkcja operatorowa, umieszczony jest on po lewej stronie słowa kluczowego **operator**). Drugim argumentem jest u2, jest to argument funkcji operatorowej. Wywołana funkcja operatorowa zwraca ułamek, który jest iloczynem ułamków **u1** i **u2**. Najprostsza definicja funkcji operatorowej może mieć postać:

```
ulamek ulamek :: operator* (ulamek x )
{ ulamek ulamek12;
ulamek12.li = li * x.li;        //oblicza licznik
ulamek12.mia = mia * x.mia;   //oblicza mianownik
return ulamek12;                //zwraca wynik mnożenia
}
```

Ta definicja jest zrozumiała, ale możemy ją uprościć. W bardziej wydajnej implementacji nie tworzymy lokalnego obiektu tymczasowego (**ulamek ulamek12**):

```
ulamek ulamek :: operator * ( ulamek x)
{  return ulamek(li * x.li, mia * x.mia);
}
```

Przeciążony operator mnożenia (*)

```
#include <iostream>
#include <conio>
using namespace std;
class ulamek
{  int li, mia;
  public:
   ulamek();
   ulamek (int, int);
   ulamek operator * (ulamek);
   void pokaz();
};

ulamek :: ulamek(): li(0), mia(1)       //licznik =0
{  }

ulamek :: ulamek ( int a, int b)
{ if (b == 0)
    { cerr << "mianownik = 0 " << endl;
      exit(EXIT_FAILURE);
    }
  li = a;    mia = b;
}
```

```
void ulamek :: pokaz()
{ cout << li << "/" << mia << endl;
}

ulamek ulamek :: operator * ( ulamek x)
{  return ulamek(li * x.li, mia * x.mia);
}

int main()
{ ulamek u1(3, 4), u2(5, 6);
  cout << "ulamek 1 = ";
  u1.pokaz();
  cout << "ulamek 2 = ";
  u2.pokaz();
  ulamek wynik ;
  wynik = u1 * u2;
  cout << "iloczyn ulamkow = ";
  wynik.pokaz();
     getche();
  return 0;
}
```

Wydruk z programu ma postać:

```
ulamek 1 = 3/4
ulamek 2 = 5/6
iloczyn ulamkow = 15/24
```

Liczby zespolone

Niemal w każdym podręczniku programowania obiektowego omawiane są przeciążane operatory arytmetyczne obsługujące liczby zespolone.

Wiek XVI, który dał początek współczesnemu rozwojowi nauki, zaznaczył się silnym rozwojem algebry. Między innymi zostały w tym czasie podane wzory wyrażające pierwiastki równań stopni 3 i 4 przez współczynniki tych równań za pomocą pierwiastków drugiego i trzeciego stopnia.

Wówczas pojawiło się zjawisko paradoksalne: Rozwiązać można tymi wzorami równanie stopnia trzeciego, które ma trzy różne pierwiastki rzeczywiste, tylko wtedy, gdy umie się obliczyć . Oczywiście w zakresie liczb do tego okresu znanych pierwiastek kwadratowy z -1 nie istniał. Nie kłopocząc się tym zbytnio, niektórzy z matematyków założyli jego istnienie i nazwali go liczbą urojoną, a poprzednio znane liczby nazwali rzeczywistymi.

Wprowadzenie tych liczb w wieku XVI nie miało żadnego uzasadnienia logicznego, ani oparcia o bezpośrednią intuicję kierowaną przez zjawiska przyrodnicze. Wskutek tego powstały kontrowersje między matematykami, z których jedni używali tych liczb bez skrępowania, inni zaś zaprzeczali ich istnieniu.

Zwolennicy istnienia tych liczb działali nimi tak jak liczbami rzeczywistymi, dodając, odejmując, mnożąc i dzieląc. Oznaczali przez i przyjmując , że .

Swobodnie dodając i mnożąc liczby rzeczywiste i "urojone" tworzyli nowe "liczby" a+bi, które dziś nazywamy liczbami zespolonymi.

Liczby zespolone

Arytmetyka tych liczb nie doprowadziła do sprzeczności. W 1748 roku Euler wprowadził je do analizy w swym fundamentalnym dziele "Introductio in analysin infinitorum", nie tylko nie dochodząc do sprzeczności, lecz powodując tym istotny postęp analizy.

Wkrótce stało się jasne, że liczby zespolone - mimo że brak im uzasadnienia logicznego - są jednym z najważniejszych narzędzi matematycznych dla badań zjawisk przyrodniczych, wskutek czego używanie ich jest w tej samej mierze słuszne, co używanie liczb rzeczywistych.

Początek wieku XIX zdarł wszelką mistykę z tych liczb, gdyż przyniósł ściśle ich uzasadnienie. Pierwsze z nich - Gaussa - wykazało, że liczby zespolone są to właściwie punkty płaszczyzny euklidesowej, w której wprowadzono pewne działania zwane dodawaniem i mnożeniem punktów czyli liczb zespolonych. Drugie uzasadnienie - Hamiltona - wprowadza liczby zespolone jako pary liczb rzeczywistych, z tym że określa się specjalny sposób mnożenia i dodawania par.

Liczby zespolone

Liczby zespolone i ich interpretacja geometryczna

Liczbą zespoloną nazywamy parę uporządkowaną liczb rzeczywistych (a,b). Często taka para zapisuje się w postaci sumy

$$z = a + bi$$

gdzie .

$$i^2 = -1$$

Tą postać liczby zespolonej nazywamy postacią kanoniczną. Liczbę (rzeczywistą) a nazywamy częścią rzeczywistą, zaś liczbę b częścią urojoną liczby zespolonej z. Część rzeczywista oznaczamy Re z, a część urojoną symbolem Im z, mamy więc:

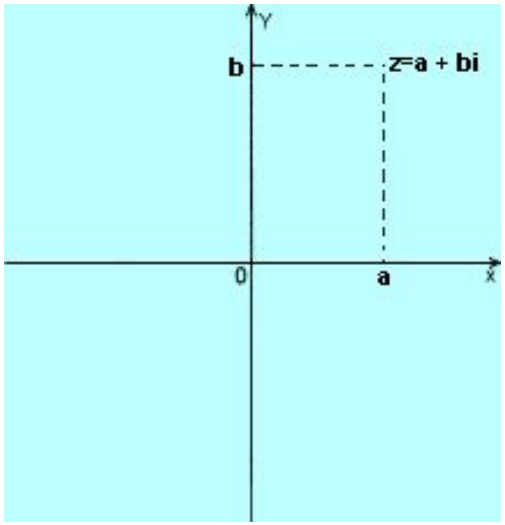
Re z = a

Im z = b.

Liczby zespolone postaci a + 0i zapisujemy jako a i utożsamiamy z liczbami rzeczywistymi. Liczba zespolona jest równa zero, wtedy i tylko wtedy, gdy Re z = 0 i Im z = 0. Zauważmy również, że kolejność liter w zapisie nie gra roli:

a + bi = a + ib = bi + a = ib + a.

Liczby zespolone interpretujemy geometrycznie jako punkty płaszczyzny. Liczbie zespolonej a + bi odpowiada punkt o współrzędnych (a,b) płaszczyzny zaopatrzonej w prostokątny układ współrzędnych. Punktom osi OX odpowiadają liczby rzeczywiste. Płaszczyznę, na której umieściliśmy liczby zespolone, nazywamy płaszczyzną Gaussa.



Przeciążenie operatorów arytmetycznych

Natomiast modułem liczby zespolonej $z = a + bi$ nazywamy liczbę

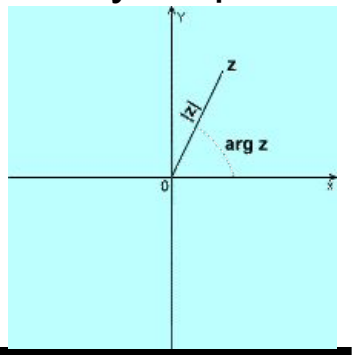
$$|z| = \sqrt{a^2 + b^2}$$

Postać trygonometryczna liczby zespolonej
Liczbę zespoloną możemy przedstawić w postaci trygonometrycznej:

$$z = a + bi = |z|(\cos\varphi + i\sin\varphi)$$

Liczbę |z| nazywamy modułem , a kąt skierowany ϕ (dokładniej jego miarę) argumentem liczby i oznaczamy arg z. Wartość argumentu liczby z czyli ϕ określamy na podstawie wartości funkcji cosinus i sinus dla , które są dane wzorami:

$$\sin\varphi = \frac{b}{|z|} \qquad \cos\varphi = \frac{a}{|z|}$$



Przeciążenie operatorów arytmetycznych

Dodajemy, odejmujemy i mnożymy liczby zespolone tak, jak wyrażenia algebraiczne pamiętając, że $i^2 = -1$. Tak więc:

$$z_1 + z_2 = (a+c) + (b+d)i$$

$$z_1 - z_2 = (a-c) + (b-d)i$$

$$z_1 \cdot z_2 = (ac-bd) + (ad+bc)i$$

$$\frac{z_1}{z_2} = \frac{(ac+bd)}{|z_2|^2} + \frac{(bc-ad)}{|z_2|^2}i$$

Pokażemy , jak można zrealizować zadanie wykonywania operacji arytmetycznych na liczbach zespolonych. Zrealizujemy nasze zadanie drogą dydaktyczną!

Przeciążenie operatorów arytmetycznych

```
#include <iostream>

using namespace std;
struct complex{
    double real, imag;
};

int main()     //dodawanie 2 liczb urojonych
{   complex a,b,z;
    a.real = 10;
    a.imag = 20;
    b.real = 15;
    b.imag = 25;
    z.real = a.real + b.real;
    z.imag = a.imag + b.imag;
    cout << "suma (a + b) = "
         << z.real<<" + i" <<z.imag<<endl;
    return 0;
}
```

Wynik:

suma (a + b) = 25 + i45

Pokazany program realizuje zadanie dobrze, ale nie jest to przykład dobrego programowania

Przeciążenie operatorów arytmetycznych

```
#include <iostream>
using namespace std;
struct complex{
    double real, imag;
};
complex dodaj(complex &a, complex &b){
    complex c;
    c.real = a.real + b.real;
    c.imag = a.imag +b.imag;
    return c;
};
int main()     //dodawanie 2 liczb urojonych
{   complex a,b,z;
    a.real = 1;
    a.imag = 2;
    b.real = 3;
    b.imag = 4;
    z = dodaj(a,b);
    cout << "suma (a + b) = "
         << z.real<<" + i" <<z.imag<<endl;
    return 0;
}
```

Wynik:

suma (a + b) = 4 + i6

Jest to lepsze kodowanie, dzięki funkcji globalnej
Ideałem będzie chyba zapis :

z = a + b;

Osiągamy taki efekt dzięki przeciążeniu operatora dodawania.

Przeciążenie operatorów arytmetycznych

```
#include <iostream>
using namespace std;
struct complex{
    double real, imag;
};
complex operator +(complex &a, complex &b){
    complex c;
    c.real = a.real + b.real;
    c.imag = a.imag +b.imag;
    return c;
};

int main()     //dodawanie 2 liczb urojonych
{   complex a,b,z;
    a.real = 11;     a.imag = 22;
    b.real = 33;     b.imag = 44;
    z = a + b;
    cout << "suma (a + b) = "
         << z.real<<" + i" <<z.imag<<endl;
    return 0;
}
```

Wynik:

suma (a + b) = 44 + i66

Zastosowano tzw. przeciążanie operatora.
Dzięki temu mamy bardzo dobre kodowanie

Przeciążenie operatorów arytmetycznych

```
#include <iostream>
using namespace std;
struct complex{
    double real, imag;
};
complex operator +(complex &a, complex &b){
    complex c;
    c.real = a.real + b.real;
    c.imag = a.imag +b.imag;
    return c;
};
void drukuj(complex &a) {
    cout << a.real<<" + i" <<a.imag<<endl;
};
int main()     //dodawanie 2 liczb urojonych
{   complex a,b,z;
    a.real = 1.1;     a.imag = 2.2;
    b.real = 3.3;     b.imag = 4.4;
    z = a + b;
    cout<<"suma = ";  drukuj(z) ;
    return 0;
}
```

Wynik:

suma = 4.4 + i6.6

Ten program ma dodaną funkcje wydruku wyniku.

Przeciążenie operatorów arytmetycznych

W pokazanych przykładach do realizacji zadania dodawania liczb zespolonych korzystaliśmy ze struktury, dostęp do danych był publicznych. Ideałem jest ochrona danych , a dostęp do nich powinien być realizowany za pomocą funkcji dostępowych. W kolejnym programie realizujemy zadanie dodawania liczb zespolonych wykorzystując odpowiednio zaprojektowaną klasę.

```
class complex
{   double real, imag;                              //dane prywatne
  public:
    complex(double r, double i) {real =r; imag = i;};       //konstruktor
    complex operator +(complex &b)            //przeciążany operator dodawania
      { return complex(real + b.real,  imag + b.imag);
      };
    void operator + ()                                  //  wydruk liczby, chytra konstrukcja!
      { cout<<real<<"   "<<imag<<endl;
      };
};
```

W instrukcji operator+ , funkcja operatorowa wymaga tylko jednego argumentu, jest to prawy operand, lewy operand przejmowany jest przez domniemanie:

```
z = a + b;
```

Operand a wywołuje operand b. Dzieje się tak, gdy funkcja operatorowa jest metodą klasy!!!

Przeciążenie operatorów arytmetycznych

W pokazanym przykładzie wykorzystaliśmy dość nietypowe rozwiązanie przeciążyliśmy operator dodawania tak, aby jego wywołanie spowodowało wydruk danych. Klasycznie do tego celu wykorzystuje się albo odpowiednią funkcję albo przeciążony operator <<.

```
void pokaz_zespolona (&x) {
cout << x.real <<"   " << x.imag << endl; }

ostream & operator << (ostream &os, nazwa_klasy &ob)
{ os << ob.x <<"     "<<ob.y <<endl;
 return os;
}
```

Druga wersja jest polecana (funkcja operatorowa jest funkcją typu „friend".

Pierwszym argumentem funkcji operator << jest referencja do obiektu typu ostream. Oznacza to, że os musi być strumieniem wyjściowym. Do drugiego argumentu ob , przesyła się w wywołaniu obiekt (adres) typu nazwa_klasy, który będzie wyprowadzany na standardowe wyjście.

Nasz ciekawostka to:

```
void operator + ()              // wydruk liczby
{ cout<<real<<"   "<<imag<<endl;
};
```

Przeciążenie operatorów arytmetycznych

```
#include <iostream>
using namespace std;
class complex
{   double real, imag;
  public:
    complex(double r, double i) {real =r; imag = i;};
    complex operator +(complex &b)
      { return complex(real + b.real,  imag + b.imag);
      };
    void operator + ()                  // wydruk liczby
      { cout<<real<<"   "<<imag<<endl;
      };
};
int main()     //dodawanie 2 liczb urojonych
{   complex a(1,0),b(2,3),z(0,0);
    cout <<"  liczba 1 =   ";
    +a;          //zlecenie wydruku wartości a!!!
    cout <<"  liczba 2 =   ";
    +b;
    cout <<"suma liczb =   ";
    z = a + b;
    +z;
   return 0;
}
```

Wynik:

```
    liczba 1 =   1   0
    liczba 2 =   2   3
suma liczb =   3   3
```

Przeciążenie operatorów arytmetycznych

```
#include <iostream>
#include <complex>          // biblioteka standardowa
using namespace std;
typedef complex<double> dcmplx;

int main(){

  dcmplx a(5.,6.),b;

  cout << "Enter b: ";
  cin >> b;

  cout << "a = " << a << "\n";
  cout << "b = " << b << "\n";

  cout << "a + b = " << a + b << "\n";
  cout << "a * b = " << a * b << "\n";
  cout << "a / b = " << a / b << "\n";
  cout << "|a| = "   << abs(a) << "\n";
  cout << "complex conjugate of a = " << conj(a) << "\n";
  cout << "norm of a = " << norm(a) << "\n";
  cout << "abs of a = " << abs(a) << "\n";
  cout << "exp(a) = " << exp(a) << "\n";
}
```

W pakiecie mamy bibliotekę complex,
W pełni obsługuje operacje na liczbach zespolonych. Należy z niej korzystać.

Wynik:

```
Enter b: (2,3)
a = (5,6)
b = (2,3)
a + b = (7,9)
a * b = (-8,27)
a / b = (2.15385,-0.230769)
|a| = 7.81025
complex conjugate of a = (5,-6)
norm of a = 61
abs of a = 7.81025
```

Przeciążenie operatora ==

Funkcje operatorowe są bardzo przydatne. Rozważmy przeciążany operator równości (==).

Implementację tej funkcji operatorowej zilustrujemy aplikacją w której porównujemy dwie daty - albo są równe, albo są nierówne.

Aby porównać dwie daty możemy wybrać operator równości aby go przeciążyć, tak aby wykonał żądane zadanie. Nasza funkcja może mieć nazwę ***operator ==.*** Ta funkcja operatorowa pobiera dwie daty (dwa obiekty) porównuje je, a następnie produkuje wynik : albo daty są równe, albo nierówne. Możemy założyć, że tworzymy klasę o nazwie ***data*** , która obsłuży obiekty typu *data*. W naszym przykładzie może być to funkcja typu bool. Możemy zatem mieć deklarację :

```
bool operator ==(data &)
```

Kolejny slajd pokazuje implementację naszego zadania.

Przeciążenie operatora ==

```
#include <iostream>
using namespace std;
class data{
    int dzien, miesiac, rok;
  public:
    data(int = 1,int = 1, int = 2000 ); //konstruktor
    bool operator==(data &); //funkcja operatorowa
};
data::data(int dd, int mm, int rr) {
    dzien = dd;
    miesiac = mm;
    rok = rr;
}

bool data::operator ==(data &dat){
    return(dzien == dat.dzien && miesiac == dat.miesiac && rok == dat.rok);
}

int main () {
   data a(4,1,2002), b(5,5, 2010), c(4,1,2002);
    if(a == c) cout <<"daty a i c sa rowne"<<endl;
    else
            cout <<"daty a i c sa nierowne"<< endl;
    if(a == b) cout <<"daty a i b sa rowne"<<endl;
    else
            cout <<"daty a i b sa nierowne"<< endl;
  return 0;
}
```

Wynik:

daty a i c sa rowne
daty a i b sa nierowne

Funkcja operatorowa w postaci niezależnej funkcji

Funkcja operatorowa może być implementowana jako ***funkcja składowa*** klasy albo jako ***zwykła funkcja globalna***. Należy pamiętać, że ***zwykła funkcja nie ma dostępu bezpośredniego do danych prywatnych***. Jeżeli chcemy, aby funkcja globalna miała dostęp do danych prywatnych klasy musimy funkcję operatorową zadeklarować jako ***funkcję zaprzyjaźnioną*** z klasą. Zanim pokażemy przykład z funkcją operatorową w postaci funkcji zaprzyjaźnionej, omówimy program, dzięki któremu wykonamy mnożenie ułamków korzystając z funkcji zaprzyjaźnionych (***nie użyjemy przeciążenia operatora***.). Klasa **ulamek** ma następującą postać:

```
class ulamek
{    int li, mia;
  public:
     ulamek (int, int);
     friend int iloczyn_Li ( ulamek &, ulamek &);
     friend int iloczyn_Mi  ( ulamek &, ulamek &);
     void pokaz();
};
```

Mnożenie ułamków w postaci niezależnej funkcji

```
#include <iostream>
#include <conio>
using namespace std;
class ulamek
{  int li, mia;
  public:
   ulamek (int, int);            //konstruktor
   friend int iloczyn_Li ( ulamek &, ulamek &);
   friend int iloczyn_Mi  ( ulamek &, ulamek &);
   void pokaz();
};

ulamek :: ulamek ( int a, int b)
{ if (b == 0)
    { cerr << "mianownik = 0 " << endl;
      exit(EXIT_FAILURE);
    }
  li = a;    mia = b;
}
```

```
void ulamek :: pokaz()
{ cout << li << "/" << mia << endl; }

int iloczyn_Li (ulamek &x, ulamek &y)
{  return (x.li * y.li);
}
int iloczyn_Mi (ulamek &x, ulamek &y)
{  return (x.mia * y.mia);
}
int main()
{ ulamek u1(3, 4), u2(5, 6);
  int licznik, mianownik;
  cout << "ulamek 1 = ";
  u1.pokaz();
  cout << "ulamek 2 = ";
  u2.pokaz();
  licznik = iloczyn_Li(u1, u2);
  mianownik = iloczyn_Mi (u1, u2);
  ulamek u3(licznik, mianownik);
  cout << "iloczyn ulamkow = ";
  u3.pokaz();
  getche();
  return 0;
}
```

Wydruk z programu ma postać:
ulamek 1 = 3/4
ulamek 2 = 5/6
iloczyn ulamkow = 15/24

Funkcja operatorowa w postaci niezależnej funkcji

W definicji klasy **ulamek** mamy dane **li** i **mia** (są to wartości odpowiednio licznika i mianownika), dwie funkcje składowe (konstruktor i funkcja **pokaz()**) oraz dwie funkcje zaprzyjaźnione. Iloczyn dwóch ułamków jest nowym ułamkiem – jego licznik jest równy iloczynowi liczników a mianownik jest równy iloczynowi mianowników. Do obliczenia iloczynów użyjemy funkcji zaprzyjaźnionych:

```
friend int iloczyn_Li ( ulamek &, ulamek &);
friend int iloczyn_Mi  ( ulamek &, ulamek &);
```

Implementacja tych funkcji ma postać:

```
int iloczyn_Li (ulamek &x, ulamek &y)
{  return (x.li * y.li);       }

int iloczyn_Mi (ulamek &x, ulamek &y)
{  return (x.mia * y.mia);    }
```

Należy zwrócić uwagę, że argumentami funkcji są referencje obiektów.

Funkcja operatorowa w postaci niezależnej funkcji

W funkcji testującej tworzymy dwie zmienne pomocnicze **licznik** i **mianownik** oraz dwa ułamki **u1** i **u2**:

```
ulamek u1(3, 4), u2(5, 6);
int licznik, mianownik;
```

a następnie wywołujemy funkcje zaprzyjaźnione:

```
licznik = iloczyn_Li(u1, u2);
mianownik = iloczyn_Mi (u1, u2);
```

dzięki którym obliczamy iloczyn liczników ułamków **u1** i **u2** oraz iloczyn mianowników tych ułamków. Mając obliczony nowy licznik i nowy mianownik tworzymy ułamek **u3** będący iloczynem ułamków **u1** i **u2**:

```
ulamek u3(licznik, mianownik);
```

Analizując powyższy przykład widzimy, że obliczanie iloczynu ułamków przy pomocy funkcji zaprzyjaźnionych wymaga skomplikowanych wywołań tych funkcji.

Mnożenie ułamków – funkcja operatorowa

W języku C++ operatory mogą być przeciążane także przy pomocy funkcji, które nie są składowymi klasy. Można stosować zwykłe ***funkcje globalne*** a także ***funkcje zaprzyjaźnione***.

Aby daną funkcję zadeklarować jako funkcję zaprzyjaźnioną z konkretną klasą, należy wstawić prototyp takiej funkcji do wnętrza definicji klasy (tak, jakby to była metoda danej klasy) i poprzedzić ten prototyp słowem kluczowym **friend**.

W zastosowaniach taka funkcja będzie traktowana jakby była metodą należącą do danej klasy. Jak pamiętamy, funkcje zaprzyjaźnione mają dostęp do danych prywatnych klasy.

W pokazanym przykładzie funkcja operatorowa jest funkcją zaprzyjaźnioną. Ponieważ do funkcji zaprzyjaźnionych ***nie jest przekazywany*** wskaźnik **this**, zaprzyjaźniona funkcja operatorowa wymaga przekazania operandów w sposób jawny.

Wobec tego funkcja operatorowa jednoargumentowa wymaga przekazania jednego parametru, funkcja operatorowa dwuargumentowa wymaga przekazania dwóch argumentów.

Należy pamiętać o kolejności przekazywanych argumentów. W funkcji operatorowej zaprzyjaźnionej lewy operand jest przekazywany jako pierwszy, a prawy jako drugi. Do przeciążania operatorów najczęściej wykorzystuje się funkcje zaprzyjaźnione, ponieważ funkcje zaprzyjaźnione są bardziej elastyczne w porównaniu z funkcjami składowymi. ***Składowe funkcje operatorowe wymagają zgodności typów operandów, natomiast w przypadku funkcji zaprzyjaźnionych nie jest wymagane, aby lewy operand koniecznie był obiektem klasy.*** Stąd wynika użyteczność zaprzyjaźnionych funkcji operatorowych - możemy mieszać typy operandów, co jest przydatne, jeżeli chcemy stosować w wyrażeniach obiekty i dane numeryczne.

Mnożenie ułamków – funkcja operatorowa zaprzyjaźniona

Pokażemy, że w znaczący sposób możemy uprościć program wprowadzając ***przeciążony operator mnożenia*** jako funkcję operatorową zaprzyjaźnioną z klasą. W kolejnym deklaracja klasy jest następująca:

```
class ulamek
{    int li, mia;
  public:
     ulamek ();                          //konstruktor
     ulamek (int, int);            //konstruktor
     friend ulamek operator*(ulamek, ulamek);
     void pokaz();
};
```

Deklaracja klasy **ulamek** zawiera deklaracje dwóch danych prywatnych **li** i **mia**, dwóch konstruktorów, jednej funkcji składowej **pokaz()** oraz operatorowej funkcji zaprzyjaźnionej. Należy pamiętać o konstruktorze bezparametrowym, bez jego jawnej definicji kompilacja naszego programu nie powiedzie się (kompilator C++ Borland 5). Deklaracja funkcji zaprzyjaźnionej ma postać:

```
friend ulamek operator*(ulamek, ulamek);
```

Mnożenie ułamków – funkcja operatorowa

Implementacja przeciążenia operatora mnożenia przy pomocy funkcji zaprzyjaźnionej może mieć postać:

```
ulamek operator*(ulamek x, ulamek y)
{ ulamek u;
  u.li = x.li * y.li;
  u.mia = x.mia * y.mia;
  return u;
}
```

Przypominamy, że w definicji klasy zaprzyjaźnionej nie podajemy nazwy klasy bazowej (bo funkcja zaprzyjaźniona nie jest funkcją składową klasy) łącznie z operatorem zakresu. Funkcja operatorowa pobiera dwa argumenty typu **ulamek** i zwraca obiekt typu **ulamek**. W ciele funkcji tworzymy tymczasowy obiekt **ulamek u**.
W funkcji testującej tworzymy trzy obiekty typu **ulamek**:

```
ulamek u1(3, 4), u2(5, 6), u3;
```

Przy pomocy prostej instrukcji:

```
u3 = u1 * u2;
```

mnożymy dwa ułamki, wykorzystując przeciążony operator mnożenia (*).

Mnożenie ułamków – funkcja operatorowa

```
#include <iostream>
#include <conio>
using namespace std;
class ulamek
{  int li, mia;
  public:
   ulamek ();                    //konstruktor
   ulamek (int, int);    //konstruktor
   friend ulamek operator*(ulamek, ulamek);
   void pokaz();
};
ulamek :: ulamek ( int a, int b)
{ if (b == 0)
    { cerr << "mianownik = 0 " << endl;
      exit(EXIT_FAILURE);
    }
  li = a;    mia = b;
}
ulamek :: ulamek ():li(0),mia(1)
{ }
```

```
void ulamek :: pokaz()
{ cout << li << "/" << mia << endl; }

ulamek operator*(ulamek x, ulamek y)
{ ulamek u;
  u.li = x.li * y.li;
  u.mia = x.mia * y.mia;
  return u;
}
int main()
{ ulamek u1(3, 4), u2(5, 6), u3;
  cout << "ulamek 1 = ";
  u1.pokaz();
  cout << "ulamek 2 = ";
  u2.pokaz();
  u3 = u1 * u2;
  cout << "iloczyn ulamkow = ";
  u3.pokaz();
    getche();
    return 0;
}
```

Wydruk z programu ma postać:
ulamek 1 = 3/4
ulamek 2 = 5/6
iloczyn ulamkow = 15/24

Przeciążenie operatora [ ]

W języku C++ operator [ ] jest operatorem binarnym stąd postać funkcji operatorowej:

```
typ nazwa:: operator [ ] ()
        {  ciało funkcji }
```

```
#include <iostream>
using namespace std;
class tab
{   int t[4];
  public :
    tab(int x0,int x1, int x2, int x3)  //konstruktor
    {  t[0]=x0;
       t[1]=x1;
       t[2]=x2;
       t[3]=x3;
    }
    int operator [] (int i) {return t[i]; }        //przeciazany operator
};
int main()
{ tab t1(2,3,5,6);
    cout << "srednia = "<< (float)(t1[0]+t1[1]+t1[2]+t1[3])/4 << endl;
    cout << "pierwszy element = "<<t1[0]<<endl;
  return 0;
}
```

Przeciążony operator []() zwraca wartość elementu tablicy wskazanego przez indeks przekazany jako parametr

Wynik:

srednia = 4
pierwszy element = 2

Zaprzyjaźnione funkcje operatorowe

Mimo zalet związanych ze stosowaniem operatorowych funkcji zaprzyjaźnionych należy pamiętać o szeregu ograniczeń. Podczas przeciążania operatorów inkrementacji i dekrementacji konieczne jest stosowanie parametrów referencyjnych.

Kolejnym ograniczenie jest fakt, że za pomocą funkcji zaprzyjaźnionych nie można przeciążać następujących operatorów:

operator przypisani (**=**)

operator odwołania do elementu tablicy (**[ ]**)

operator odniesienia do składowej klasy (**->**)

operator **( )**

Przeciążanie operatorów równości i nierówności (operatorowa funkcja składowa)

Zagadnienie przeciążania operatorów równości i nierówności zilustrujemy przykładami. Załóżmy, że chcemy przeładować operatory równości (**==**) i nierówności (**!=**). Dla prostoty weźmy klasę reprezentującą trójwymiarowy wektor. Należy przeciążyć operator **==** oraz **!=** w ten sposób, aby można było ustalić równość i nierówność dwóch wektorów. Przeciążanie należy zrealizować:

- jako funkcje składowe
- jako funkcje zaprzyjaźnione

Definicja klasy może mieć postać:

```
class wektor3d
{   double x, y, z;
  public:
      wektor3d (double w1=0.0, double w2=0.0, double w3 = 0.0)
        {     x = w1;    y = w2;    z = w3;
        }
      int operator == ( wektor3d );
      int operator !=  ( wektor3d );
};
```

Przeciążanie operatorów równości i nierówności (operatorowa funkcja składowa)

Jak widać w klasie **wektor3d** zadeklarowane są dwie funkcje składowe:

```
int operator == (wektor3d);
int operator != (wektor3d);
```

Każda z nich ma jeden argument typu **wektor3d** i jeden argument domniemany – **this**. Implementacji funkcji **operator==** ma postać:

```
int wektor3d::operator == ( wektor3d v )
{      if ( (v.x == x) && (v.y == y) && (v.z == z) ) return 1;
                                                    else return 0;
}
```

Składowe dwóch wektorów są kolejno porównywane, gdy wszystkie są równe zwracana jest wartość 1, w przeciwnym przypadku zwracana jest wartość 0. Należy zwrócić uwagę, że w funkcji implementującej przeciążony operator nierówności wykorzystano przeciążony ***operator równości*** !

Ta funkcja ma postać:

```
int wektor3d::operator != (wektor3d v)
{      return ! ( (*this) == v) ;
}
```

Przeciążanie operatorów równości i nierówności (operatorowa funkcja zaprzyjaźniona)

Kolejny przykładzie pokazuje przeciążania operatorów równości i nierówności wykorzystując funkcje zaprzyjaźnione W tym przypadku klasa ma postać:

```
class wektor3d
{ double x, y, z;
   public:
      wektor3d (double w1=0.0, double w2=0.0, double w3 = 0.0)
         {    x = w1;  y = w2;  z = w3;       }
      friend int operator == (wektor3d, wektor3d);
      friend int operator != (wektor3d, wektor3d);
 };
```

W klasie **wektor3d** zadeklarowano dwie operatorowe funkcje zaprzyjaźnione o nazwach **operator ==** i **operator !=**:

```
      friend int operator == (wektor3d, wektor3d);
           friend int operator != (wektor3d, wektor3d);
```

Przeciążanie operatorów równości i nierówności (operatorowa funkcja zaprzyjaźniona)

Będą one otrzymywać po dwa argumenty typu **wektor3d**. Implementacja tych funkcji ma postać:

```
 operator == (wektor3d v, wektor3d w)
{   if ( (v.x == w.x) && (v.y == w.y) && (v.z == w.z) ) return 1;
                                          else return 0;
}
operator != (wektor3d v, wektor3d w)
   {      return ! ( v == w) ;
   }
```

Przeciążanie operatorów równości i nierówności (operatorowa funkcja zaprzyjaźniona)

Widzimy, że operator przeciążony może być zdefiniowany jako funkcja składowa i jako funkcja globalna (lub zaprzyjaźniona). Wobec tego musimy odpowiedzieć na pytanie, jakie są kryteria wyboru implementacji przeciążania operatora. Prawdę mówiąc ***nie ma*** generalnej zasady, wszystko zależy od zastosowania przeciążonego operatora.

Jeżeli operator modyfikuje operandy to powinien być zdefiniowany jako funkcja składowa klasy. Przykładem są tu takie operatory jak: = , += , -= , *= , ++ , itp.

Jeżeli operator nie modyfikuje swoich operandów to należy go definiować jako funkcję globalną lub zaprzyjaźnioną.

Przykładem są tu takie operatory jak: + , - , == , & , itp.

Przeciążanie operatorów przypisania

Operator przypisania jest szczególnym operatorem, ponieważ w przypadku, gdy nie zostanie przeciążony, jest on ***definiowany*** przez kompilator. Tak więc, operacja przypisania jednego obiektu drugiemu ***jest zawsze wykonalna***. Ilustruje ten fakt kolejny program. W naszym przykładzie została zaimplementowana klasa **data**:

```
class data
{
 private:
     int dzien;
     int miesiac;
     int rok;
public:
    data (int = 1, int = 1, int = 2004 ) ;      //konstruktor
    void pokaz(void);                           // funkcja skladowa, pokazuje date
};
```

Ta klasa nie zawiera funkcji operatorowej przypisania.

Przeciążanie operatorów przypisania

```
int main()
{ data d1(13,3,2003), d2( 15, 5, 2004);
  cout << "\n pierwsza data, d1: " ;
  d1.pokaz();
  cout << "\n    druga data, d2: ";
  d2.pokaz();
  d1 = d2;
  cout << "\npo podstawieniu d1: ";
  d1.pokaz();
  cout << endl;
    getche();
    return 0;
}
```

W funkcji głównej main( ) mamy instrukcję:

d1 = d2;

co oznacz, że odpowiednie pola obiektu d2 są przypisane polom obiektu d1. Ten typ przypisania nosi nazwę przypisania danych składowych (***memberwise assignment***).

Jeżeli mamy proste przypisanie, tak jak to pokazano powyżej, wygenerowane przez kompilator przeciążenie jest wystarczające, jednak w bardziej skomplikowanych przypadkach (np. gdy chcemy mieć wielokrotne przypisanie typu: **d1 = d2 = d3**) musimy zaprojektować odpowiednią funkcję operatorową.

Przeciążanie operatorów przypisania

Deklaracja prostego operatora przypisania może mieć postać:

```
void operator = (nazwa_klasy & )
```

Słowo kluczowe void wskazuje, że przypisanie nie zwróci żadnej wartości, napis operator = wskazuje, że przeciążamy operator przypisania, a nazwa klasy i znak & wewnątrz nawiasów okrągłych wskazują, że argumentem operatora jest ***referencja*** do klasy. Na przykład, aby zadeklarować operator przypisania dla naszej klasy data, możemy użyć deklaracji:

```
void operator = ( data &);
```

Implementacja funkcji operatorowej może mieć postać:

```
void Data :: operator= (Data & d)
{  dzien = d.dzien;
   miesiac = d.miesiac;
   rok = d.rok;
}
```

Przeciążanie operatorów przypisania

W definicji operatora zastosowano referencję. W tej definicji **d** jest zdefiniowane jako referencja do klasy **Data**. W ciele funkcji operatorowej składowa **dzien** obiektu **d** jest przypisana składowej **dzien** aktualnego obiektu:

```
dzien = d.dzien;
```

Ta sama operacja powtórzona jest dla składowych m**iesiac** i **rok**. Przypisanie typu:

```
a.operator=(b);
```

może być zastosowane do wywołania przeciążonego operatora przypisania i przypisania wartości składowych obiektu **b** do obiektu **a.** W tej sytuacji zapis **a.operator=(b)** może być zastąpiony wygodnym zapisem **a = b;.**

Funkcja operatora przypisania zaprezentowana w tym przykładzie, aczkolwiek poprawna, nie obsłuży poprawnie próby przypisania wielokrotnego postaci:

```
a = b = c;
```

Dzieje się tak, ponieważ powyższy zapis interpretowany jest jako:

```
a = ( b = c );
```

Przeciążanie operatorów przypisania

Zgodnie z definicją, nasza funkcja operatorowa nie zwraca żadnej wartości, wobec tego po wykonaniu przypisania **b = c**, żadna wartość nie będzie zwrócona i nic nie możemy przypisać do **a**. W celu wykonywania wielokrotnego przypisania, potrzebna jest funkcja operatorowa zwracająca referencję do klasy swojego typu.

Przy pomocy wskaźnika **this** zmienimy implementację operatorowej funkcji przypisania tak, aby możliwe było ***wielokrotne przypisanie***.

Zasadniczą sprawą jest zaprojektowanie funkcji **operator=** tak, aby mogła zwrócić wartość typu **Data**. Prototyp takiej funkcji może mieć postać:

```
Data operator= ( const Data & ) ;
```

Zastosowaliśmy specyfikator **const** do parametru funkcji, aby mieć pewność, że ten operand nie będzie zmieniony przez funkcję.

Implementacja nowej funkcji operatorowej może mieć postać:

```
Data Data :: operator=(const Data &d)
{  dzien = d.dzien;
   miesiac = d.miesiac;
   rok = d.rok;
   return *this;  }
```

Przeciążanie operatorów przypisania

Należy pamiętać, że funkcja operatorowa przypisania musi być ***funkcją składową klasy***, nie może być funkcja zaprzyjaźnioną.

W przypadku przypisania takiego jak **b = c** , (równoważna forma **b.operator=(c)**), wywołana funkcja zmienia dane składowe obiektu **b** na dane obiektu **c** i zwraca nową wartość obiektu **b**. Taka operacja umożliwia wielokrotne przypisanie typu **a = b = c**.

Implementacja przeciążonego operatora przypisania i jego zastosowanie pokazano na kolejnym wydruku.

```
class Data
{   int dzien, miesiac,rok;
 public:
   Data(int, int, int);   //konstruktor
  void pokaz();
  Data operator=(const Data &);
};
```

Przeciążanie operatora wstawiania do strumienia (<<)

Wiele zastosowań praktycznych mają funkcje realizujące przeciążanie operatorów wstawiania danych (**<<**) do strumienia i operatorów pobierania danych (**>>**) ze strumienia. Jak wiemy, w języku C++ operator << jest operatorem powodującym przesuwanie bitów o żądaną liczbę pozycji. Fakt, że możemy użyć tego operatora na przykład przy wyświetlaniu wartości:

```
int x = 13;
cout << x;
```

zawdzięczamy technice przeciążania operatorów. Powyższy zapis ma następującą interpretację:

```
cout.operator<<(x);
```

cout jest egzemplarzem obiektu klasy **ostream**, klasa ta jest zawarta w bibliotece standardowej. Operator << jest zdefiniowany w klasie **ostream**, a co więcej jest on przeciążony w ten sposób, że możemy go wykorzystywać dla wszystkich wbudowanych typów danych.

Możliwe jest również takie przedefiniowanie tego operatora, aby można było wyświetlać dane typów zdefiniowanych przez użytkownika. Ma to duże znaczenie praktyczne

Przeciążanie operatora wstawiania do strumienia (<<)

W kolejnym przykładzie pokażemy jak można przeciążyć operator wstawiania, aby można było wyświetlić dane obiektu przy pomocy jednej instrukcji.

Klasyczna postać definicji operatora wstawiania (<<) jest następująca:

```
ostream & operator<< (ostream & os, nazwa_klasy & ob)
      {
  //instrukcje
          return os;
}
```

Zdefiniowana funkcja jest klasy **ostream &,** czyli musi podawać referencje do obiektu klasy **ostream &.** Pierwszym argumentem funkcji **operator<<()** jest referencja do obiektu typu **ostream**. Oznacza to, że **os** musi być strumieniem wyjściowym. Drugim argumentem także jest referencja, do argumentu **ob** przesyła się obiekt typu **nazwa_klasy**. Funkcja **operator<<** zawsze zwraca referencję do swojego pierwszego argumentu, to znaczy do strumienia wyjściowego **os.** Zaprojektowana funkcja operatorowa musi być zaprzyjaźniona z klasą **nazwa_klasy**, jeżeli chce mieć dostęp do składowych chronionych klasy.

Przeciążanie operatora wstawiania do strumienia (<<)

Jeżeli mamy następującą klasę:

```
class punkt
{   int x,y;
public:
    punkt(int a, int b) { x = a; y = b;}   //konstruktor
    friend ostream& operator<< (ostream &, punkt & );
};
```

to funkcja operatorowa może mieć postać:

```
ostream & operator<< (ostream & os, punkt & ob)
{  os << "x = " << ob.x << endl;
   os << "y = " << ob.y << endl;
   return os;
}
```

Krótki program ilustrujący omawiane zagadnienie pokazany jest na wydruku.

Przeciążanie operatora wstawiania do strumienia (<<)

```
#include <iostream>
#include <conio>
using namespace std;
class punkt
{ int x,y;
  public:
   punkt(int a, int b) { x = a; y = b;}        //konstruktor
   friend ostream& operator<< (ostream &, punkt & );
};
ostream & operator<< (ostream & os, punkt & ob)
{  os << "x = " << ob.x << endl;
   os << "y = " << ob.y << endl;
   return os;
}
int main()
{  punkt p1(5, 15);
   cout << p1;    // piękne!!!!
   cout << "wywolanie jawne: \n";
   operator<<(cout, p1);
return 0;
}
```

Wydruk z programu ma postać:

```
x = 5
y = 15
wywolanie jawne:
x = 5
y = 15
```

Zadania

Z1.Korzystając z klasy , opracować funkcję operatorową mnożenia do mnożenia, dodawania i odejmowania dwóch licz zespolonych.

Funkcja operatorowa jest metodą klasy.

Z2.Korzystając z klasy , opracować funkcję operatorową mnożenia do mnożenia, dodawania i dzielenia dwóch ułamkow.

Funkcja operatorowa jest metodą klasy.

KONIEC