

Programowanie obiektowe

Konwersje

Problem konwersji (1)

- Klasa opisująca liczby zespolone:

```
class zesp {  
public:  
    double re, im;  
    zesp(double r, double i) : re(r), im(i) {};  
};
```

- Definiujemy funkcję dodającą dwie liczby zespolone:

```
zesp add(zesp x, zesp y) {  
    zesp w;  
    w.re = x.re + y.re;  
    w.im = x.im + y.im;  
    return w;  
}
```

- Co zrobić, gdy chcemy dodać liczbę rzeczywistą (która jest szczególnym przypadkiem liczby zespolonej) do liczby zespolonej? Czy trzeba definiować nowe funkcje?

Problem konwersji (2)

- Funkcja **add** oczekuje dwóch argumentów typu **zesp** – nie można jej wywołać z argumentem typu **zesp** i **float**.
- Należałoby zdefiniować jeszcze po dwie dodatkowe funkcje dla każdego typu zmiennoprzecinkowego:
 - `zesp add(double x, zesp y);`
 - `zesp add(zesp x, double y);`
- Można by było tego uniknąć, gdyby kompilator wiedział, jak przekształcać liczby typu **double** na typ **zesp** – wtedy wystarczyłaby tylko jedna definicja funkcji **add**, działająca na argumentach typu **zesp**.

Konwersja obiektu typu X na typ Y może być zdefiniowana przez użytkownika.

Konstruktor jako konwerter (1)

Konstruktor przyjmujący jeden argument określa konwersję od typu tego argumentu do typu klasy, do której należy.

- Aby zdefiniować w klasie **zesp** konwersję od typu **double** należy zdefiniować taki konstruktor:

```
zesp(double x) : re(x), im(0.0) {}
```

- Ten sam efekt można było otrzymać prościej, definiując:

```
zesp(double r, double i = 0.0) : re(r), im(i) {}
```

- Konstruktor taki będzie wywoływany **niejawnie** zawsze, gdy funkcja oczekuje na argument typu **zesp**, a dostanie **double**, np. wywołanie:

```
add(liczba_zesp, 5.0);
```

kompilator zrozumie jako: `add(liczba_zesp, zesp(5.0));`

- Teraz wystarczy już tylko jedna definicja funkcji add:

```
zesp add(zesp x, zesp y);
```

Konstruktor jako konwerter (2)

- Typ, z którego dokonujemy konwersji może być typem dowolnym, np. zdefiniowanym przez użytkownika.
- Konwersja z klasy **X** do **Y**:
 - konstruktor definiujemy w klasie **Y** (utworzy on obiekt klasy **Y**),
 - konstruktor ten musi znać wewnętrzną strukturę klasy **X**, zatem:
 - istotne składowe klasy **X** muszą być publiczne albo
 - klasa **X** musi deklarować przyjaźń z tym konstruktorem albo
 - klasa **X** musi udostępniać publiczne funkcje składowe umożliwiające konstruktorowi dostęp do jej prywatnych składowych.

Konstruktor jako konwerter (3)

```
class zesp1;
class zesp2 { ////////////////////////////////////////////
    double re, im;
public:
    zesp2(double r=0.0, double i=0.0) : re(r), im(i) {}
                                   // konwersja double -> zesp2
    zesp2(zesp1);                  // konwersja zesp1 -> zesp2
};
class zesp1 { ////////////////////////////////////////////
    float mod, arg;
    friend zesp2::zesp2(zesp1);
public:
    zesp1(float m, float a) : mod(m), arg(a) {}
}; ////////////////////////////////////////////
zesp2::zesp2(zesp1 x) {
    re = x.mod * cos(x.arg); // mod i arg - pryw.
    im = x.mod * sin(x.arg); // skład. w zesp1
}
```

Operator konwersji (1)

- Sytuacja odwrotna – chcemy dokonać konwersji typu zdefiniowanego przez użytkownika na typ wbudowany:

Funkcja konwertująca – funkcja składowa klasy **X**, która ma nazwę: **X::operator T()**, gdzie **T** jest nazwą typu, na który dokonujemy konwersji (typu wbudowanego lub zdefiniowanego przez użytkownika!).

- Przykłady:

```
zesp2::operator double() {  
    return re;        // konwersja zesp2 -> double  
}  
  
zesp1::operator float() {  
    return mod;        // konwersja zesp1 -> float  
}
```

Operator konwersji (2)

- Funkcja konwertująca:
 - musi być funkcją składową klasy (posiada wskaźnik **this**, który konwertuje);
 - nie ma określonego typu zwracanego – zawsze zwraca taki typ, jak się sama nazywa (instrukcja **return**);
 - ma pustą listę argumentów (nie można jej przeładować);
 - jest dziedziczona,
 - może być funkcją wirtualną.
- Przykłady:

```
void fun(float) ;  
//...  
zesp1 x(1.0, 2.0), y(-0.5, 1.3) ;  
fun(x); // niejawna konwersja zesp1 -> float  
  
float f = (float) y; // jawna konwersja ...
```

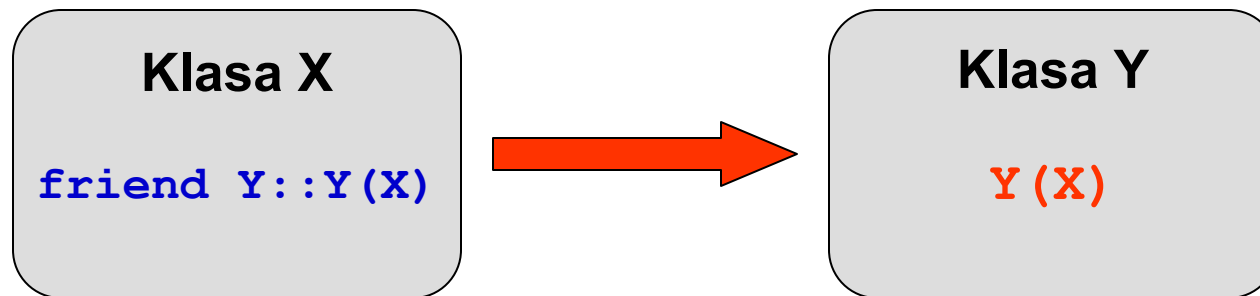

Operator konwersji (3)

- Funkcja konwertująca może dokonać konwersji na typ wbudowany lub zdefiniowany przez użytkownika.
- Przykład: konwersja z typu **zesp2** do typu **zesp1**:

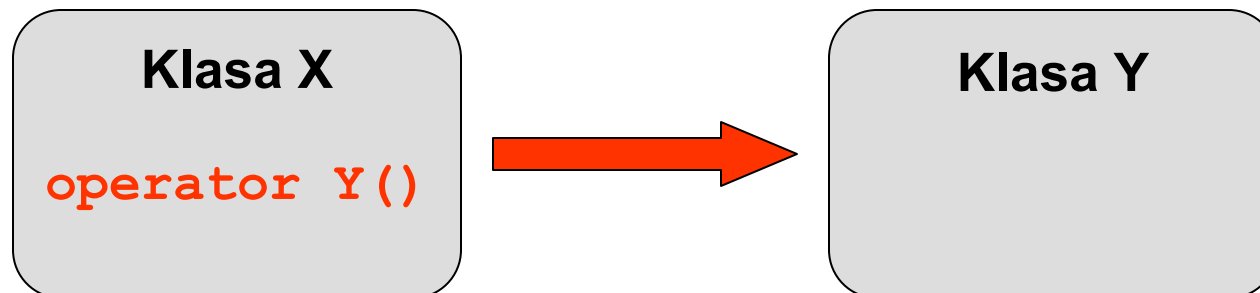
```
class zesp2 {
    double re, im;
public:
    zesp2(double r=0.0, double i=0.0) : re(r), im(i) {}
                                   // konwersja double -> zesp2
    operator double() { return re; }
                                   // konwersja zesp2 -> double
    operator zesp1(); // konwersja zesp1 -> zesp2
};
//...
zesp2::operator zesp1() {
    zesp1 w(sqrt(re * re + im * im), atan(im / re));
    return w;
}
```

Porównanie

- Do przekształcenia jednego typu w drugi mamy dwa narzędzia:
 - konstruktor jednoargumentowy:



- funkcję konwertującą (operator konwersji).



Który wariant wybrać?

- Jeśli wybierzemy obie wersje:
 - w wypadku, gdy kompilator będzie musiał dokonać niejawnej konwersji, nie będzie wiedział, którą wersję wybrać \Rightarrow błąd!
 - można wtedy używać tylko konwersji jawnej, tzn. wywoływać `explicit` konstruktor lub operator konwersji.
- Warto jednak korzystać z konwersji niejawnych!
- A zatem trzeba dla danego przypadku konwersji dokonać wyboru metody konwersji.

Wady wersji z konstruktorem

- Nie można zdefiniować konstruktora dla typu wbudowanego.
- Nie można napisać konstruktora dla klasy bibliotecznej, tzn. dostarczonej przez innego użytkownika.
- Jeśli piszemy konstruktor dla własnej klasy, to i tak musi on korzystać z informacji o obcej klasie.
- Przy konstruktorze konwertującym nie można korzystać z żadnych konwersji standardowych.
- Konstruktor konwertującego nie dziedziczy się.

Wskazówki

- Konwersja obiektu klasy **X** na typ wbudowany – tylko operatorem konwersji.
- Konwersja obiektu klasy **X** na obiekt klasy **Y**:
 - lepiej posłużyć się operatorem konwersji – nie wymaga on ingerencji w klasę **Y**,
 - jeśli klasa **X** jest niedostępna (np. klasa biblioteczna), ale istotne jej składowe są dostępne (bezpośrednio lub za pomocą publicznych funkcji składowych) – należy posłużyć się konstruktorem konwertującym.

Konwersje niejawne

- Konwersje są wywoływane niejawnie, gdy równocześnie istnieje możliwość jednoznacznej konwersji mogącej usunąć niedopasowanie oraz:
 - w wywołaniu funkcji występuje niezgodność argumentów aktualnych, np. `(objX, 2.0)` z argumentami formalnymi np. `(X, X)`,
 - przy zwracaniu wartości funkcji, obiekt zwracany (po `return`) ma inny typ niż deklarowany dla rezultatu tej funkcji (w nagłówku funkcji),
 - w obecności operatorów – konwertowane są jeden lub dwa operandy wyrażenia,
 - w wyrażeniach:
`X objX;`
`if(objX) ... while(objX) ... switch(objX) ...`
określana jest wartość `(int) objX`,
 - w wyrażeniach inicjalizujących, np. `float k = objX.`

Konwersje jawne

- Konwersja jawna może mieć dwie formy zapisu:

```
zesp2 z(1.0, 2.0);  
double x;
```

- forma rzutu:
- forma wywołania funkcji:

```
x = (double) z;  
x = double(z);
```

- Obie te formy robią to samo, jednak nie są całkowicie identyczne.
- W większości przypadków można posłużyć się oboma zapisami.
- Są jednak wypadki, kiedy jeden z zapisów jest preferowany.

Konwersje jawne – wywołanie funkcji

- Konwersja z klasy `zesp1` `zesp2`:

- konstruktor konwertujący:

```
zesp2(zesp1 z, double q = 0.0);
```

- dostępny jest także operator konwersji:

```
zesp1::operator zesp2();
```

- Obie formy rzutowania są teraz dopuszczalne.
- Jeżeli jednak do rzutowania użyć takiego jawnego wywołania konstruktora:

```
zesp2 z = zesp2(z1, 3.0);
```

to możliwa jest tylko ta forma (z wywołanie funkcji).
- W drugiej formie – rzutowania – nie da się przesłać drugiego argumentu.

Konwersje jawne – rzutowanie

- Forma rzutowania jest najczęściej preferowana, gdy typ, na który zamieniamy, ma zbyt skomplikowaną składnię.

```
X::operator char*(); // rzutow. na wsk. do char  
  
X obj;  
char* napis = (char *) obj; // rzutowanie
```

- Forma funkcyjna byłaby nielegalna:

```
napis = char*(obj); // błąd!
```

- Można to ominąć definiując nową nazwę dla typu:

```
typedef char* tekst;  
napis = tekst(obj);
```

Niezupełne dopasowanie (1)

- Dwie przeładowane funkcje:

```
fun(float) ;  
fun(X) ;
```

- Operator konwersji:

```
X::operator float() ;
```

- Która z powyższych wersji funkcji zostanie wywołana?

```
X obj ;      fun(obj) ;
```

Zostanie wywołana funkcja **fun(X)** dlatego, że pasowała ona **dokładnie** do wywołania – bez konieczności stosowania jakiejkolwiek konwersji.

Niezupełne dopasowanie (2)

- Funkcja: `fun(float) ;`
- Operator konwersji: `X::operator int() ;`
- Czy poniższe wywołanie jest poprawne?
`X obj ;`
`fun(obj) ;`
- Tak, ponieważ nastąpi taka konwersja (kaskadowa):

$X \Rightarrow \text{int} \Rightarrow \text{float}$

W konwersji kaskadowej konwersja zdefiniowana przez użytkownika może wystąpić tylko **jednokrotnie!**

- Tzn. nie wykona się **niejawnie** konwersja postaci:

$X \Rightarrow Y \Rightarrow \text{float}$

Niezupełne dopasowanie (3)

- Dwie funkcje:
`fun(float) ;`
`fun(X) ;`
- Konwersja: `int ⇒ X`.
- Która z powyższych funkcji zostanie wywołana przy takim zapisie:

```
fun(1) ; // dla int
```

Wykona się funkcja `fun(float)`, bo wystarczyła konwersja standardowa `int ⇒ float`.

Konwersje zdefiniowane przez użytkownika są używane niejawnie **w dodatku** do konwersji standardowych!

Niezupełne dopasowanie (4)

- Dwie funkcje: `fun(int) ; fun(float) ;`
- Dwie konwersje: `x ⇒ int, x ⇒ float.`
- Która wersja zostanie wywołana?

```
x obj ;  
fun(obj) ;
```

- Żadna – dwuznaczność \Rightarrow błąd!
- Dwie funkcje: `fun(long) ; fun(float) ;`
- Teraz dwuznaczności nie ma – wywoła się funkcja `fun(float)`: jest bowiem operator konwersji, który zamienia obiekt klasy `x` na obiekt typu `float`, bez pośredniej konwersji.

Niezupełne dopasowanie (5)

- Dwa operatory konwersji:

```
X::operator int();    // prywatny  
X::operator float();  // publiczny
```

- Dwie funkcje:

```
fun(int); fun(float);
```

- Wywołanie:

```
X obj;  
fun(obj);
```

jest nielegalne, bo występuje dwuznaczność.

Zawsze najpierw rozsądza się sprawę dwuznaczności,
a dopiero potem sprawdza dostęp!

Niezupełne dopasowanie (6)

- Konwersja z typu **x** na **y** realizowana jest przez:
 - operator konwertujący w klasie **x**,
 - konstruktor konwertujący w klasie **y**.
- Zapis błędny – dwuznaczność:

```
x objx; Y objy;  
objy = (Y) objx;  
objy = Y(objx);
```

- Zapis poprawny:

```
objy = objx.operator Y();
```

Rady

- Poprawny schemat konwersji (*zbieżny*):

$$A \Rightarrow X, \quad B \Rightarrow X, \quad C \Rightarrow X.$$

- Niepoprawny schemat konwersji (*rozbieżny*):

$$X \Rightarrow A, \quad X \Rightarrow B, \quad X \Rightarrow C.$$

- Nie mnożyć konwersji ponad potrzebę.
- Starać się, by klasa miała jeden operator konwersji do innego typu!
- Jeśli będą potrzebne inne konwersje dla tej klasy, napisz funkcję składową, która się tym będzie zajmować.
- Operator konwersji między dwoma klasami powinien zamieniać w stronę obiektu o prostszej strukturze!