Rozdział 10

Przeciążanie operatorów

10.1 Definicje

Język C++ umożliwia przeciążanie operatora, tzn. zmianę jego znaczenia na potrzeby danej klasy. W tym celu definiujemy funkcję o nazwie:

```
operator op
```

gdzie `op` jest nazwą konkretnego operatora. Funkcja ta może być metodą, czyli funkcją składową klasy lub też zwykłą funkcją. Ponadto musi mieć co najmniej jeden argument danej klasy, co uniemożliwia zamianę działania operatorów dla wbudowanych typów danych takich jak int, float itp.

Dla przykładu rozważmy klasę Wektor zdefiniowaną w rozdziale 8. W niniejszym rozdziale utworzymy klasę WektorR pochodną od klasy Wektor, w której przeciążymy kilka operatorów. Oczywiście, można też bezpośrednio zmienić definicję klasy Wektor, ale przy okazji pokażemy możliwość wprowadzania różnych uzupełnień bez zmiany danej klasy. Programowanie obiektowe to doskonałe narzędzie do tego celu. Jak wiemy, wystarczy zdefiniować klasę pochodną.

W klasie Wektor była podana metoda:

```
void Wektor::Dodaj(Wektor b)
{
   int i;
   for (i=0; i<liczba; i++)
```

```
      pocz[i] += b.pocz[i];
}
```

Metodę tę wywołujemy następująco:

```
a.Dodaj(b);
```

gdzie a i b są obiektami klasy Wektor.

Przypomnijmy sobie, jaką operację realizuje metoda Dodaj(). Otóż metodę Dodaj() można by zapisać następująco:

```
void Wektor::Dodaj(Wektor b)
{
   int i;
   for (i=0; i<this->liczba; i++)
      this->pocz[i] += b.pocz[i];
}
```

Zmiana jaka nastąpiła, to jawne dodanie wskaźnika **this** wskazującego obiekt, na rzecz którego dana metoda została wywołana. Jeżeli zatem napiszemy:

```
a.Dodaj(b);
```

to wskaźnik **this** wskazuje na obiekt a. Realizowana operacja zaś to:

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

A w skróconym zapisie:

```
a += b;
```

Warto się zastanowić, czy nie byłoby sensownie zastąpić metodę Dodaj() po prostu operatorem +=. Nie potrzeba wtedy pamiętać jaką operację realizuje metoda Dodaj(). W języku C++ wystarczy zdefiniować metodę o nazwie:

```
operator +=
```

i następującej treści (metodę definiujemy w klasie pochodnej):

```
void WektorR::operator += (WektorR b)
{
   for (int i=0; i<liczba; i++)
      pocz[i] += b.pocz[i];
}
```

Wskaźnik **this** nie występuje jawnie w tej metodzie, bo jak wiemy dodaje go automatycznie kompilator.

Jeżeli teraz napiszemy:

```
a+=b;
```

to zapis ten jest równoważny:

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

czyli wywołujemy metodę operator+= z argumentem `b` na rzecz obiektu `a`. Wskaźnik **this** wskazuje zatem na obiekt `a`. Z tych wyjaśnień od razu widać, że jeżeli funkcja przeciążająca dany operator występuje jako metoda, to lewy argument operacji musi być obiektem danej klasy (jest on przekazywany przez wskaźnik **this**).

Warto jeszcze wspomnieć, że nie wolno zapomnieć o dodaniu do deklaracji klasy WektorR wiersza:

```
void operator += (WektorR);
```

który deklaruje metodę przeciążającą operator +=.

Dla porównania podamy teraz funkcję przeciążającą operator +=. Funkcja ta ma postać:

```
void operator += (WektorR &a, WektorR &b)
{
   for (int i=0; i < a.liczba; i++)
      a.pocz[i] += b.pocz[i];
}
```

Parametry są przekazywane przez referencję, co pozwala na ich zmianę w wyniku działania funkcji. W przeciwnym przypadku argumenty byłyby przekazywane przez wartość (jest to dopuszczalne tylko dla parametru `b`), co uniemożliwiłoby wykonanie operacji:

```
a += b;
```

W treści funkcji korzystamy z pól prywatnych klasy WektorR, a zatem funkcja operator+= powinna być zaprzyjaźniona z klasą WektorR. Jak pamiętamy z poprzedniego rozdziału, dokonujemy tego przy pomocy deklaracji:

```
friend void operator += (WektorR &, WektorR &);
```

umieszczonej w deklaracji klasy WektorR.

Należy podkreślić, że deklaracja zaprzyjaźnienia jest potrzebna tylko wtedy, gdy funkcja ma działać na polach prywatnych lub chronionych klasy.

W języku C++ można przeciążać większość operatorów za wyjątkiem:

```
::        .* . ?:
```

Ponadto nawet po przeciążeniu są zachowane pierwotnie zdefiniowane reguły pierwszeństwa. Na przykład wyrażenie:

```
x - y / z
```

odpowiada następującemu:

```
x - (y / z)
```

bez względu na to, jakie operacje wykonują operatory - oraz /. Operator jednoargumentowy po przeciążeniu musi takim pozostać. Podobnie operator dwuargumentowy nie może zmienić liczby argumentów.

Łączność operatorów również nie może się zmienić. Na przykład wyrażenie:

```
x = y = z;
```

wykonuje się jako:

```
x = (y = z);
```

Ten sam operator może być przeciążony więcej niż raz, ale za każdym razem z innym zestawem parametrów. Natomiast nie można utworzyć jednocześnie metody i funkcji z tymi samymi parametrami.

10.2 Operator jako metoda czy funkcja

Jak już wiemy, funkcja definiująca operator może być metodą jak i funkcją. Jeżeli jest to metoda, to ma zawsze o jeden parametr mniej niż funkcja. Wynika to stąd, że metoda ma niejawny wskaźnik **this**, wskazujący obiekt, na rzecz którego dana metoda jest aktywowana.

Warto się teraz zastanowić, którą technikę powinno się stosować. W pewnych przypadkach jest to oczywiste.

W języku C++ cztery operatory, a mianowicie: =, [ ], (), `->` muszą być definiowane jako metody.

Z drugiej strony trzeba pamiętać, że w przypadku metody lewy argument operacji musi być obiektem danej klasy. Czasami się zdarza, że lewy argument powinien być innego typu. W tym przypadku operator powinien być zdefiniowany jako funkcja. Przypomnijmy, że jeżeli chcemy, by funkcja miała dostęp do prywatnych lub chronionych składowych klasy, to powinna być zadeklarowana jako zaprzyjaźniona z daną klasą.

W pozostałych przypadkach wybór należy do programisty i do jego indywidualnych upodobań.

10.3 Operator przypisania =

Operator przypisania jest szczególnym operatorem, ponieważ w przypadku, gdy nie zostanie przeciążony, jest on definiowany przez kompilator. Operacja przypisania jednego obiektu drugiemu jest zawsze wykonalna. Niestety, analogicznie jak w przypadku konstruktora kopiującego, jeżeli w klasie istnieją wskazania na części dynamiczne, operator wygenerowany przez kompilator nie będzie działał prawidłowo. Musimy wtedy zaprojektować własny operator przypisania w analogiczny sposób jak przy przeciążaniu innych operatorów. Jeżeli chcemy dokonać przypisania:

```
a = b;
```

to funkcję przeciążajacą operator = musimy zaprojektować jako metodę (operator = jest jednym z czterech operatorów, które muszą być przeciążane przy pomocy metod). Musimy pamiętać, że na obiekt `a` będzie wskazywał wskaźnik **this**, a obiekt `b` powinien być parametrem.

Pierwsza wersja metody operator= może wyglądać następująco:

```
void WektorR::operator = (WektorR &b)
{
   // usunięcie części dynamicznej obiektu wskazywanego
   // przez wskaźnik this
   delete [liczba] pocz;

   // utworzenie tablicy dynamicznej dla obiektu
   // wskazywanego przez wskaźnik this na podstawie
   // wielkości tablicy dynamicznej obiektu b
   pocz = new int[liczba = b.liczba];

   // przepisanie danych zawartych w tablicy dynamicznej
   // obiektu b do tablicy dynamicznej obiektu wskazywanego
   // przez wskaźnik this
   for (int i=0; i<liczba; i++)
      pocz[i] = b.pocz[i];
}
```

Na wszelki wypadek przypomnijmy sobie jeszcze, że na przykład instrukcja:

```
delete [liczba] pocz;
```

jest równoważna następującej:

```
delete [liczba] this->pocz;
```

Ponadto w nowszych wersjach kompilatorów można pominąć rozmiar tablicy i napisać:

```
delete [] pocz;
```

Analizując treść metody operator= widzimy, że składa się ona z dwóch podstawowych części. Pierwsza z nich to usunięcie części dynamicznej obiektu wskazywanego przez wskaźnik **this** (dla instrukcji a=b wskaźnik **this** wskazuje na obiekt `a`). Druga część to utworzenie od początku części dynamicznej i wpisanie do niej danych na podstawie obiektu `b`. I taka jest ogólna reguła obowiązująca przy konstrukcji metody przeciążającej operator =.

Warto też zauważyć, że usunięcie części dynamicznej obiektu jest konieczne tylko wtedy, gdy:

```
liczba != b.liczba
```

Uwagę tę uwzględnimy w następnych wersjach metody.

Niestety, zaprojektowana metoda nie działa jeszcze całkowicie prawidłowo. Otóż w przypadku przypisania:

```
a = a;
```

instrukcja ta jest oczywiście równoważna:

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

W trakcie wykonania usunęlibyśmy część dynamiczną obiektu wskazywanego przez wskaźnik **this** - czyli obiektu `a` i następnie nie byłoby możliwe dokonanie przepisania danych z obiektu, który jest parametrem aktulnym metody operator=, czyli obiektu `a`. W tym przypadku najlepiej nie podejmować żadnych działań, co możemy zapewnić sprawdzając warunek:

```
if (this != &b) {
```

Operator `&` podaje adres obiektu `b`, czyli powyższa instrukcja pozwala na sprawdzenie, czy obiekt wskazywany przez wskaźnik **this** i obiekt będący parametrem formalnym to te same obiekty.

Po tej modyfikacji metoda operator= przyjmuje postać:

```
void WektorR::operator = (WektorR &b)
{
   // sprawdzenie czy obiekt wskazywany przez wskaźnik this
   // i obiekt będący parametrem to te same obiekty
   if (this != &b) {
      if (liczba != b.liczba) {
         // rózne rozmiary tablic
         delete [liczba] pocz;
         pocz = new int[liczba = b.liczba];
      }
      for (int i=0; i<liczba; i++)
         pocz[i] = b.pocz[i];
   }
}
```

Zdefiniowany operator przypisania działa prawidłowo w przypadku instrukcji typu:

```
a = b;
```

natomiast nie może działać w przypadku instrukcji:

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

Wynika to stąd, że powyższa instrukcja jest równoważna:

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

a po wykonaniu przypisania b = c metoda operator= nie zwraca żadnej wartości (jest typu void). Nie możemy przypisać żadnej wartości obiektowi `a`. Ten drobny mankament możemy łatwo naprawić zmieniając typ metody na WektorR `&` (dodaliśmy symbol referencji). Metoda powinna zatem podawać obiekt typu WektorR, a więc musimy dodać instrukcję:

```
return *this;
```

W przypadku przypisania:

```
b = c;
```

metoda podaje obiekt wskazywany przez wskaźnik **this** czyli w naszym przypadku obiekt `b`. No i teraz nic już nie stoi na przeszkodzie, aby stosować instrukcje typu:

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

A oto treść metody operator= po tej modyfikacji:

```
WektorR & WektorR::operator = (WektorR &b)
{
   if (this != &b) {
      if (liczba != b.liczba) {
         delete [liczba] pocz;
         pocz = new int[liczba = b.liczba];
      }
      for (int i=0; i<liczba; i++)
         pocz[i] = b.pocz[i];
   }
   return *this;
}
```

Warto jeszcze zapamiętać, że operator przypisania = nie może być dziedziczony i w klasie pochodnej musi być skonstruowany oddzielnie.

Na zakończenie tego podrozdziału podamy jeszcze przykład przeciążenia operatora przypisania w klasie ZbiorLiczb omawianej w rozdziale 8 (tym razem nie tworzymy klasy pochodnej i definicję operatora dodajemy bezpośrednio do klasy ZbiorLiczb). Warto teraz przypomnieć sobie konstrukcję tej klasy zwracając szczególną uwagę na sposób pamiętania zbioru (lista liczb). Poniżej jest podana treść metody operator=. Warto zwrócić uwagę na wyraźnie widoczne dwie części: jedną zaczerpniętą z destruktora klasy i drugą z konstruktora kopiującego.

```
ZbiorLiczb & ZbiorLiczb::operator = (ZbiorLiczb obiekt)
{
   // sprawdzenie czy obiekt wskazywany przez wskaźnik this
   // i obiekt będący parametrem to te same obiekty
   if (this != &obiekt) {
      // usunięcie części dynamicznej obiektu wskazywanego
      // przez wskaźnik this (instrukcje te są analogiczne
      // jak w destruktorze)
      wezel *pom;
      wezel *wsk = pierwszy;
      while(wsk) {
         pom = wsk;
         wsk = wsk->nastepny;
         delete pom;
      }

      // przepisanie danych przechowywanych w liście
      // obiektu będącego parametrem do listy obiektu
      // wskazywanego przez wskaźnik this (instrukcje te
      // są analogiczne jak w konstruktorze kopiującym)
      ile = obiekt.ile;
      wezel *stara = obiekt.pierwszy;
      wezel *nowa;

      pierwszy = NULL;
      while(stara) {
         nowa = new wezel;
         nowa->element = stara->element;
```

```
         nowa->nastepny = pierwszy;
         pierwszy = nowa;
         stara = stara->nastepny;
      }
   }
      return *this;
}
```

Nie należy jeszcze zapomnieć o dodaniu do deklaracji klasy ZbiorLiczb następującej instrukcji:

```
ZbiorLiczb & operator = (ZbiorLiczb);
```

10.4 Operator [ ]

Operatorem bardzo często przeciążanym w przypadku wykorzystywania tablic jest operator dostępu do elementu tablicy [ ]. Trzeba sobie jednak zdawać sprawę z tego, że operator ten wcale nie musi być wykorzystywany razem z tablicami. Może wykonywać zupełnie dowolną operację. Musi być jednak definiowany jako metoda czyli funkcja składowa klasy.

My wykorzystamy operator [ ] w sposób tradycyjny do indeksowania tablicy dodatkowo wraz z kontrolą czy indeks nie przekroczył dozwolonych wartości. W tym celu projektujemy następującą metodę:

```
int & WektorR::operator [] (int i)
{
   if ( i<0 ) {
      // zmiana indeksu
      i = 0;
      // zapamiętanie faktu przekroczenia zakresów
      przekroczenie = 1;
   }

   if ( i>=liczba) {
      // zmiana indeksu
      i = liczba-1;
      // zapamiętanie faktu przekroczenia zakresów
      przekroczenie = 1;
   }
```

```
    return pocz[i];
}
```

W metodzie tej sprawdzamy, czy indeks (parametr formalny) mieści się w zakresie między 0, a wartością pola `liczba`. W przeciwnym przypadku zmieniamy indeks na 0 i zapamiętujemy fakt przekroczenia w polu `przekroczenie`.

Operator [ ] może być wykorzystywany zarówno przy pobieraniu wartości z tablicy jak i przy wpisywaniu wartości do tablicy. Wynika to z tego, że wynik wykonania metody jest przekazywany przez referencję. A zatem po deklaracjach:

```
WektorR a;
int x;
```

prawidłowe będą na przykład instrukcje:

```
x = a[2];
```

oraz

```
a[1] = 100;
```

Gdyby nagłówek metody wyglądał tak:

```
int WektorR::operator [] (int i)
```

to wtedy instrukcja:

```
a[1] = 100;
```

byłaby błędna. Wynik podawany przez metodę operator[ ] jest wtedy wartością, a nie adresem zmiennej, a jak wiadomo nie można do pewnej wartości przypisać innej.

Operator [ ] zastępuje zatem znacznie mniej wygodne metody Podaj() oraz Wpisz() dostępne w klasie bazowej Wektor.

Zwróćmy uwagę, że w metodzie operator[ ] występuje instrukcja:

```
przekroczenie = 1;
```

Pole `przekroczenie` musimy umieścić w części **private** klasy WektorR i zaprojektować tak konstruktor klasy, by wpisać odpowiednią wartość początkową. Treść konstruktora jest następująca (po dwukropku jest wywoływany konstruktor klasy bazowej Wektor):

```
WektorR::WektorR(int n):Wektor(n)
{
   przekroczenie = 0;
}
```

Pomocnicza metoda Sprawdz() pozwala na stwierdzenie, czy zakres został przekroczony:

```
void WektorR::Sprawdz()
{
   if (przekroczenie)
      cout << "Zakres został przekroczony";
}
```

A oto częściowa deklaracja klasy WektorR (pełną deklarację klasy zamieścimy pod koniec rozdziału):

```
class WektorR : public Wektor
{
private:
   int przekroczenie;
public:
   WektorR(int);    // konstruktor
   int & operator [] (int i);
   void Sprawdz();
}
```

10.5 Operator ()

Operator () nie może być zdefiniowany w postaci zwykłej funkcji. Jego określenie musi mieć postać metody danej klasy. Wyjątkowość tego operatora polega na tym, że jako jedyny pozwala na przekazanie większej liczby argumentów niż dwa. Jeżeli zatem chcemy przesłać więcej niż dwa parametry, nie mamy wyboru, musimy użyć operatora (). Wykorzystanie tego operatora zilustrujemy następującym przykładem.

Przykład 10.1 W przykładzie pokażemy możliwość sprawdzania, czy w tablicy dwuwymiarowej indeksy nie przekroczyły zadanego zakresu. Zauważmy, że w tym przypadku nie możemy wykorzystać operatora [ ]. Wynika to stąd, że obecnie musimy przekazać trzy argumenty: obiekt reprezentujący tablicę oraz dwa indeksy.

Na wstępie zdefiniujemy klasę Tablica wyłącznie w celu ilustracji wykorzystania operatora (). Dlatego też w tej klasie nie zaprojektujemy konstruktora kopiującego oraz operatora przypisania. Niech to będzie ćwiczenie dla Czytelnika (są to metody podobne do zamieszczonych w klasie Wektor i WektorR). W klasie Tablica muszą być pola chronione (zostawiamy możliwość tworzenia klas pochodnych) określające liczbę wierszy oraz kolumn tablicy dwuwymiarowej oraz adres początku tablicy. A oto deklaracja klasy Tablica:

```
class Tablica
{
protected:
   int m;   // liczba wierszy
   int n;   // liczba kolumn
   int *pocz;  // adres
public:
   Tablica(int, int);    // konstruktor
   ~Tablica();           // destruktor
   int & operator() (int, int);
};
```

Przed zaprojektowaniem odpowiednich metod zastanówmy się, jak najwygodniej utworzyć tablicę dwuwymiarową. Otóż można elementy tablicy dwuwymiarowej przechowywać w tablicy jednowymiarowej (wektorze) w ten sposób, że w wektorze umieszczamy kolejno wiersze tablicy dwuwymiarowej. A zatem zadaniem konstruktora jest utworzenie wektora dynamicznego o rozmiarze m * n i przypisanie elementom tablicy wartości 0. Definicja konstruktora jest następująca:

```
Tablica::Tablica(int liczbaW, int liczbaK)
{
   m = liczbaW;
   n = liczbaK;
   pocz = new int[m*n];
```

```
   for (int i; i < m*n; i++)
      pocz[i] = 0;
}
```

Poniżej jest zamieszczony destruktor analogiczny jak dla klasy Wektor.

```
Tablica::~Tablica()
{
   delete [m*n] pocz;
}
```

No i najważniejsza metoda w tym przykładzie:

```
int & Tablica::operator() (int i, int j)
{
   // sprawdzenie poprawności zakresu pierwszego indeksu
   if ( i< 0 || i >= m)
      i = 0;

   // sprawdzenie poprawności zakresu drugiego indeksu
   if (j < 0  || j >= n)
      j = 0;

   return pocz[i * n + j];
}
```

Metoda ta zwraca referencję do elementu leżącego w i-tym wierszu i j-tej kolumnie tablicy. Element ten odpowiada i * n + j elementowi wektora wskazywanego przez `pocz`. Metoda jest typu int `&`. Jak pamiętamy z określenia operatora [ ] dla klasy Wektor umożliwia to wykorzystanie tych operatorów po lewej stronie instrukcji przypisania.

Powyższa metoda definiuje operator () o trzech argumentach. Trzech, dlatego że pierwszym jest niejawny wskaźnik **this**. Instrukcja

```
   return pocz[i * n + j];
```

może być bowiem zapisana nastepująco:

```
   return this->pocz[i * n + j];
```

gdzie wskaźnik **this** wskazuje obiekt, na rzecz którego został wywołany operator ().

Jeżeli na przykład utworzymy obiekt `t`:

```
Tablica t(2,2);
```

to operator () możemy wykorzystywać na dwa sposoby. Albo pisząc:

```
t(0,0) = 10;
```

lub

```
t.operator() (0,0) = 10;
```

Oczywiście pierwszy sposób jest znacznie wygodniejszy, natomiast w drugim wyraźnie widać, że operator () jest wywoływany na rzecz obiektu `t` wskazywanego przez wskaźnik **this**.

Trzeba tylko sobie zdawać sprawę z tego, że zapis

```
t(0,0) = 10;
```

jest różny od stosowanego dla tablic dwuwymiarowych w języku C++ i dlatego może być nieco mylący.

10.6 Operator <<

Bardzo często warto zdefiniować operator `<<` i wyprowadzać elementy wektora pisząc:

```
cout << x;
```

gdzie `x` jest obiektem interesującej nas klasy.

Oczywiście, operator `<<` jest zdefiniowany w klasie ostream (klasa ostream jest zawarta w bibliotece standardowej), a co więcej jest on przeciążony w ten sposób, że możemy go wykorzystywać dla wszystkich wbudowanych typów danych. Na przykład instrukcje:

```
int k = 0;
cout << k;
```

oznaczają, że wywołujemy operator << zdefiniowany w klasie ostream w sposób nastepujący:

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

gdzie cout jest obiektem klasy ostream a wersja operatora jest zdefiniowana dla parametru typu int. Z kolei instrukcje:

```
float x = 0;
cout << x;
```

oznaczają, że wywołujemy wersję operatora << zdefiniowaną dla parametru typu float.

Jasne jest, że klasa ostream nie ma pojęcia o istnieniu naszej klasy Wektor i na pewno w klasie ostream nie ma odpowiedniej wersji operatora <<, takiego by możliwa była sekwencja:

```
WektorR x;
cout << x;
```

Operator << musimy zatem przeciążyć, tzn. utworzyć nową wersję funkcji definiującej ten operator rozróżnialny od poprzednich na podstawie typu parametrów. Do wyboru mamy teraz dwie możliwości. Może to być funkcja lub metoda zdefiniowana w klasie WektorR. Przypomnijmy sobie teraz, że dla metody pierwszym argumentem musi być obiekt danej klasy. Pomysł, aby to była metoda klasy WektorR, odpada, ponieważ nie do przyjęcia jest zapis:

```
WektorR x;
x << cout;
```

Jest to oczywiście możliwe do zrealizowania, ale mylące. Na szczęście nie ma żadnych przeciwskazań, aby zdefiniować operator << przy pomocy funkcji. A oto przykładowa definicja tej funkcji:

```
ostream & operator<< (ostream & ob, WektorR & wek)
{
   ob << endl << "ELEMENTY WEKTORA " << endl;
   for(int i=0; i < wek.liczba; i++)
      ob << wek.pocz[i] << endl;
   return ob;
}
```

Funkcja jest klasy ostream &, czyli musi podawać referencję do obiektu klasy ostream &. Jest to potrzebne po to, by można było stosować instrukcje:

```
WektorR x,y;
cout << x << y;
```

Argumentami funkcji są obiekt klasy ostream i obiekt klasy WekorR. Oba są przekazywane przez referencję, ale to w tej chwili nie jest istotne. Natomiast bardzo ważna jest kolejność tych argumentów. Otóż przy podanej kolejności możemy napisać:

```
WektorR x;
cout << x;
```

co odpowiada wywołaniu:

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

Jest to zapis naturalny, do którego jesteśmy przyzwyczajeni.

Podkreślmy jeszcze, że operator << zawarty w poniższej instrukcji:

```
ob << wek.pocz[i] << endl;
```

jest operatorem zdefiniowanym w klasie ostream. Rozróżnienie operatora zdefiniowanego przez nas i operatora z klasy ostream nie nastręcza żadnych trudności. Nasz operator ma bowiem dwa argumenty typu: ostream i WektorR, natomiast operator z klasy ostream ma argumenty typu: ostream oraz int.

No i na koniec bardzo ważna sprawa. Zaprojektowana przez nas funkcja musi być zaprzyjaźniona z klasą WektorR. Jest to konieczne, ponieważ musi mieć ona dostęp do składników chronionych klasy. Do deklaracji klasy WektorR musimy zatem dodać deklarację zaprzyjaźnienia:

```
friend ostream & operator<< (ostream &, WektorR &);
```

10.7 Niektóre inne operatory

Podamy teraz jeszcze kilka przykładów operatorów zaprojektowanych dla klasy WektorR.

Najpierw zmodyfikujemy nieco definicję operatora += omawianego w podrozdziale 10.1. Po pierwsze sprawdzimy, czy operacja ma sens. Chodzi o to, że w przypadku, gdy drugi wektor ma mniejszą długość niż pierwszy, to działanie jest nieokreślone, ponieważ sumujemy zgodnie z liczbą elementów pierwszego wektora. Po drugie warto umożliwić wielokrotną operację rodzaju:

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

W tym celu, podobnie jak w przypadku operatora przypisania, wystarczy zażądać, aby funkcja podawała obiekt wskazywany przez wskaźnik **this**. A oto zmodyfikowana treść metody operator+=.

```
WektorR & WektorR::operator += (WektorR b)
{
   int i;
   if (liczba <= b.liczba)
      for (i=0; i<liczba; i++)
         pocz[i] += b.pocz[i];
   else
      cout << "Przy dodawaniu wektorów drugi wektor"
      << " ma mniejszą długość"
      << endl << "Operacji nie wykonano" << endl;
   return *this;
}
```

Dwuargumentowy operator -= można zaprojektować analogicznie jak operator +=. Natomiast teraz podamy definicję jednoargumentowego operatora -. Zwróćmy uwagę, że argument ten jest przekazywany poprzez wskaźnik **this** i dlatego w metodzie operator- brak jest parametrów formalnych.

```
void WektorR ::operator -()
{
   for (int i=0; i<liczba; i++)
      pocz[i] = -pocz[i];
}
```

Zdefiniowany operator może być wykorzystywany następująco:

```
-a;
```

Treść metody definiującej operator == może być następująca:

```
int WektorR:: operator == (WektorR &b)
{
   int i;
   // jeśli liczba elementów jest różna
   if (liczba  != b.liczba)
      return 0;
   else
      for (i=0; i<liczba; i++)
         if (pocz[i] != b.pocz[i])
            return 0;
   return 1;
}
```

Podamy jeszcze treści metod definiujących operatory ++ oraz --.

```
void WektorR:: operator ++ ()
{
   for (int i=0; i<liczba; i++)
      pocz[i]++;
}

void WektorR:: operator -- ()
{
   for (int i=0; i<liczba; i++)
      pocz[i]--;
}
```

Oba zdefiniowane wyżej operatory są podane w wersji przedrostkowej. Warto wiedzieć, że od wydania 3.0 języka C++ można zdefiniować wersje przyrostkowe tych operatorów poprzez dodanie drugiego argumentu typu int.

Przypomnijmy, że w klasie mogą być zdefiniowane przez kompilator tylko dwa operatory = oraz &. Wszystkie inne powinny być w miarę potrzeby definiowane przez programistę.

Zwróćmy uwagę, że wszystkie zdefiniowane operatory mają podobne znaczenie jak dla typów wbudowanych. I tak być powinno. Określając jakiś operator, nie powinniśmy łamać dotychczasowych przyzwczajeń.

A oto deklaracja obiecanej wcześniej klasy WektorR pochodnej od klasy Wektor.

```
class WektorR : public Wektor
{
   friend ostream & operator<< (ostream &, WektorR &);
private:
   int przekroczenie;
public:
   WektorR(int);
   int  & operator [] (int i);
   void Sprawdz();
   WektorR & operator += (WektorR);
   void operator *= (int);
   WektorR & operator = (WektorR &);
   void operator -();
   WektorR & operator -= (WektorR);
   int operator == (WektorR);
   void operator ++ ();
   void operator -- ();
};
```

Popatrzmy jeszcze, w jaki sposób można zastąpić wywołanie metody Dodaj() zaprojektowanej w klasie ZbiorLiczb operatorem. Oczywiście najpierw trzeba się zastanowić, który operator zastosować. W tym przypadku nie ma bezpośredniej odpowiedniości pomiędzy symbolem operatora a jego funkcją. Wydaje się, że najbardziej odpowiedni będzie operator <. Stosując ten operator zamiast pisać:

```
zb.Dodaj(3);
```

można wykonać instrukcję:

```
zb < 3;
```

Metoda definiująca operator < ma postać (bazuje ona na realizacji metody Dodaj()):

```
void ZbiorLiczb::operator < (int nowy)
{
   if ( !Nalezy(nowy)) {
      wezel *wskN = new wezel;
      wskN -> nastepny = pierwszy;
```

```
      wskN -> element = nowy;
      pierwszy = wskN;
      ile++;
   }
}
```

Jak zwykle nie możemy zapomnieć o dodaniu do deklaracji klasy ZbiorLiczb deklaracji:

```
   void operator < (int nowy);
```

10.8 Wnioski

Podsumujmy teraz informacje zawarte w tym rozdziale. Otóż projektując nową klasę, trzeba się dokładnie zastanowić, które operatory warto zdefiniować. Jeżeli klasa ma składowe dynamiczne, to bezwzględnie trzeba zaprojektować operator przypisania =, ponieważ wygenerowany przez kompilator nie zapewni prawidłowego działania. Natomiast przy definicji innych operatorów należy się kierować poprawą czytelności programu. Jeżeli operator realizuje funkcję, która nie ma związku ze zwyczajowym znaczeniem operatora, to lepiej do tego celu zaprojektować metody o jasnych nazwach.