Referencje

Referencje można uznać za niejawny wskaźnik. To może być użyte:

- jako parametr funkcji,
- jako wartość zwracana przez funkcję,
- jako zmienna referencyjna.

- Referencje jako parametr funkcji. Parametry można przekazać do funkcji na trzy sposoby:

  - przez wartość - kopia argumentu jest przekazywana do funkcji.
  - przez wskaźnik - przekazywany jest wskaźnik do argumentu.
  - przez referencję – przekazywany jest niejawny wskaźnik (referencje) do argumentu.

- Wskaźnik do argumentu w C++ można przekazać jawnie (tak jak w C) i niejawnie - można użyć parametru referencyjnego.

Aby zadeklarować parametr referencyjny przed nazwą argumentu formalnego, należy umieścić &.

W ciele funkcji nie piszemy * przed nazwą argumentu formalnego (operator wskazania pośredniego). Instrukcja n = 100; oznacza, że liczba 100 jest przypisana do rzeczywistego argumentu, który zastępuje formalny argument n podczas wywoływania funkcji.
Podczas wywoływania funkcji nie umieszczamy & przed rzeczywistym parametrem.

Parametr referencyjny nie jest wskaźnikiem - arytmetyka wskaźników nie działa tutaj:

```
int k = 100;
fun(k);
void fun(int &n)
{
      n++;            //increases n by one.
      n = n+20;       // the value of n will be
                      // increased by 20
}
```

```
int k = 100;
fun(&k);
void fun(int *n)
{
      n++;             //  n points to the address n+sizeof(int)
      n = n+20;        //n points to the address
                       //n + 20*sizeof(int)
      n = n – 21;      //points to the k
      *n = *n+20;      //increasing the variable with the
                       //address n by 20
}
```

Przekazywanie obiektów do funkcji przez referencje

Przekazanie obiektu jako argumentu (przez wartość) tworzy kopię tego obiektu, więc odbywa się wywołanie destruktora przy wyjściu z funkcji - możliwe skutki uboczne, czas na wywołanie funkcji jest stosunkowo długi ze względu na kopiowanie argumentów. Zmiany danych obiektu dokonywane w funkcji odbywają się w kopii, nie wpływając na stan danych oryginalnego obiektu.

Przekazanie obiektu przez referencję nie tworzy jego kopii. Po zakończeniu funkcji obiekt przekazany przez parametr nie jest niszczony - destruktor nie jest wywoływany. Przekazywanie jest szybsze niż przekazywanie przez wartość (nie tworzy się kopia obiektu). Zmiany w danych obiektu dokonane w funkcji, po zakończeniu działania funkcji, pojawiają się w pierwotnym obiekcie. Składnia jest taka sama jak przy przekazywaniu wartości:

```
name_of_reference_to_object.variable
name_of_reference_to_object.class_method(list of arguments);
```

Example W16.

Funkcja zwraca referencje

Funkcje mogą zwracać referencje. Umożliwia to umieszczenie tej funkcji po lewej stronie operatora przypisania, a także jest korzystne przy przeciążaniu operatorów.

Example W17.

Przeciążenie operatorów

Przeciążanie operatorów pozwala na użycie tej samej składni dla obiektów klas i struktur, jak dla standardowych typów obiektów.

Operator można przeciążyć, tworząc operator-funkcje. Operator-funkcja definiuje, jakie operacje operator powinien wykonać na obiektach określonej klasy. Słowo kluczowe *operator* służy do tworzenia funkcji operatora. Operator-funkcja może być składową klasy (a może nie być). Najczęściej funkcja-operator, która nie jest składową klasy, jest funkcją zaprzyjaźnioną do tej klasy. Funkcja - operator jest funkcją konwencjonalną, ale ma specyficzny interfejs wywoływania. Na przykład, w linii kodu

```
my_class a(….), b(….);
a = b;
```

kompilator automatycznie wywołuje operator =, jeśli klasa my_class zawiera przeciążenie funkcji - operatora = . W przeciwnym razie (jeśli nie zawiera), kompilator zastosuje kopię bitową.

Funkcja - operator – składowa klasy

```
typ  class_name :: operator  #  (list_of_arguments)
{
      //function body
}
```

Często typem wartości zwracanej przez funkcję - operator jest klasa, metodą której występuje ta funkcja. Znak # wskazuje działanie operatora, który przeciążamy.

Ograniczenia dotyczące przeciążenia operatorów:

- Priorytet operatorów nie może być zmieniany.
- Nie wolno zmieniać liczby operandów operatora.
- Operatory, które nie mogą być przeciążone: . :: -> ?
- Operatory - funkcje nie mogą mieć parametrów domyślnych.

Operator — funkcje, za wyjątkiem operatora =, dziedziczą się z klasy pochodnej.

W klasie pochodnej można przeciążyć dowolny operator, nawet te operatory, które były już przeciążone w klasie bazowej.

Aby poprawnie przeciążyć operator, musimy jasno zrozumieć następujące:

- Ile operandów ma operator.
- Dane jakiego typu zwraca funkcja - operator.
- Jaki jest typ każdego operandu.
- Czy możliwe jest przeciążenie operatora jako funkcji składowej klasy, albo jest konieczne przeciążenie go jako osobnej funkcji i zaprzyjaźnienie się do klasy.

Przeciążenie operatorów binarnych.

- Taki operator ma dwa operandy.
- Funkcja – operator składowa klasy ma tylko jeden argument – referencja lub wskaźnik do operandu prawostronnego. Operand lewostronny jest przekazywany niejawnie – przez wskaźnik *this*.

Przeciążenie operatora przypisania

Jest to operator binarny, ma dwa operandy – lewostronny i prawostronny.

Example:

```
coord & coord::operator = (const coord &right)
/*================================================
Overload of the assignment operator: returns the reference to object
================================================*/
{
   //ciało funkcji-operatora
   return *this;
}
```

Operator zwraca referencje do obiektu, ponieważ lewostronny operand występuje po lewej stronie operatora przypisania (L-value). Po przypisaniu lewostronny operand ulegnie zmianie. Funkcja – operator zwraca * *this* – pobiera obiekt przez wskaźnik i zwraca referencje do tego obiektu. Nadaje to możliwość wykorzystać łańcuch a = b = c;
Tu najpierw obiekt c jest przepisywany do objectu b (b = c), a dalej – obiekt b – do obiektu a (a = b) – prawostronna łączność operatora = .

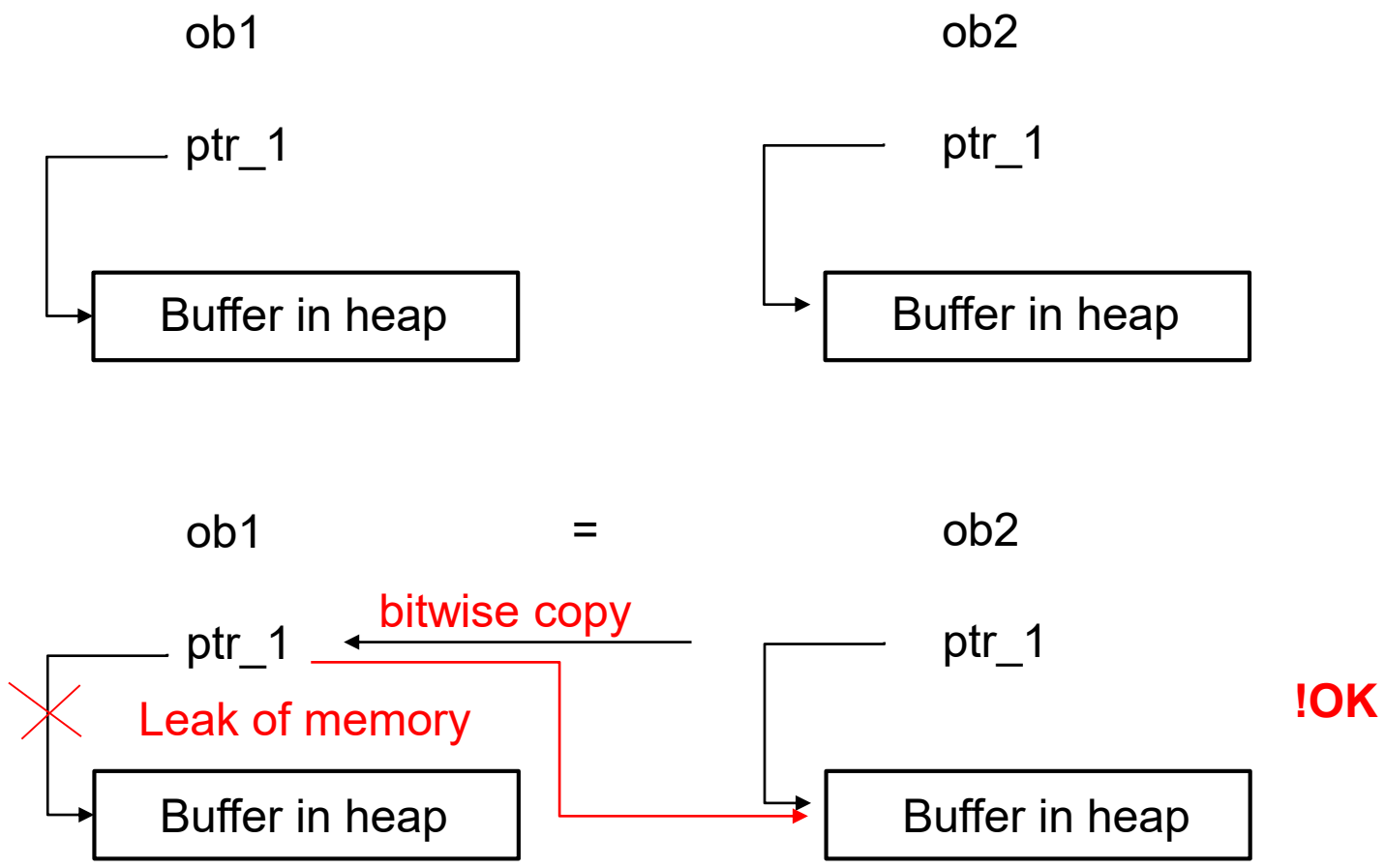
Argumentem operatora = jest referencja do operandu prawostronnego, ponieważ przekazywanie przez referencje nie powoduje tworzenia kopii obiektu, a po zakończeniu – wywołania destruktora. Nadaje to możliwość znacznie przyspieszyć wywołanie tego operatora oraz uniknąć efektów ubocznych.

Przypisanie obiektów

- Jeśli typy dwóch obiektów są takie same, jeden obiekt można przypisać do drugiego. Domyślnie przypisanie jednego obiektu do drugiego oznacza, że dane obiektu po prawej stronie będą kopiowane bit po bicie do obiektu po lewej.

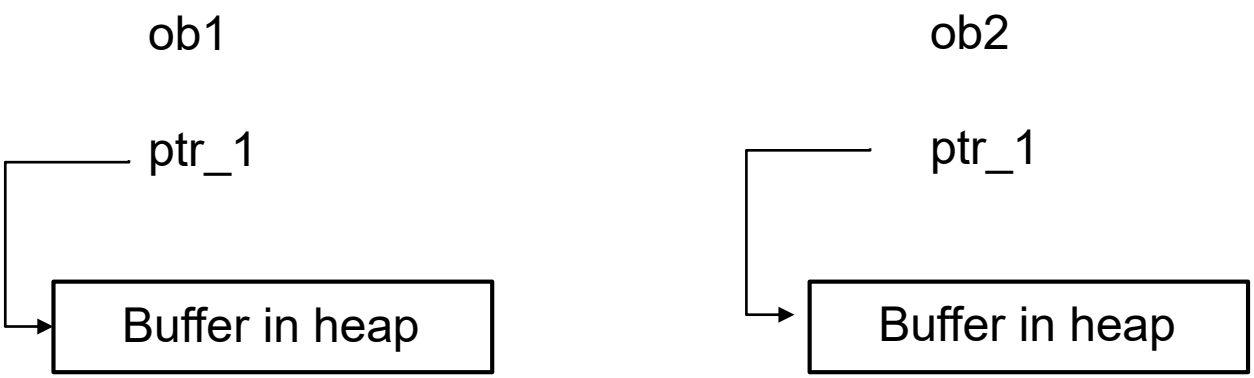
```
class myclass
{
        int i;
public:
        myclass(int ii) : i(ii) { }  //It is similar to: myclass(int ii) { i = ii; }
        myclass( ) : i(0) { }
        int get_i() { return i; }
};

int main()
{
        myclass ob1(10), ob2;
        ob2 = ob1;
        cout << ob1.get_i() << ”\t” << ob2.get_i() << endl;
……..
}

        Output:
        10      10
```

➢ Jeśli dane klasy zawierają wskaźniki do obiektów dowolnego typu, przypisanie domyślne może spowodować błąd.

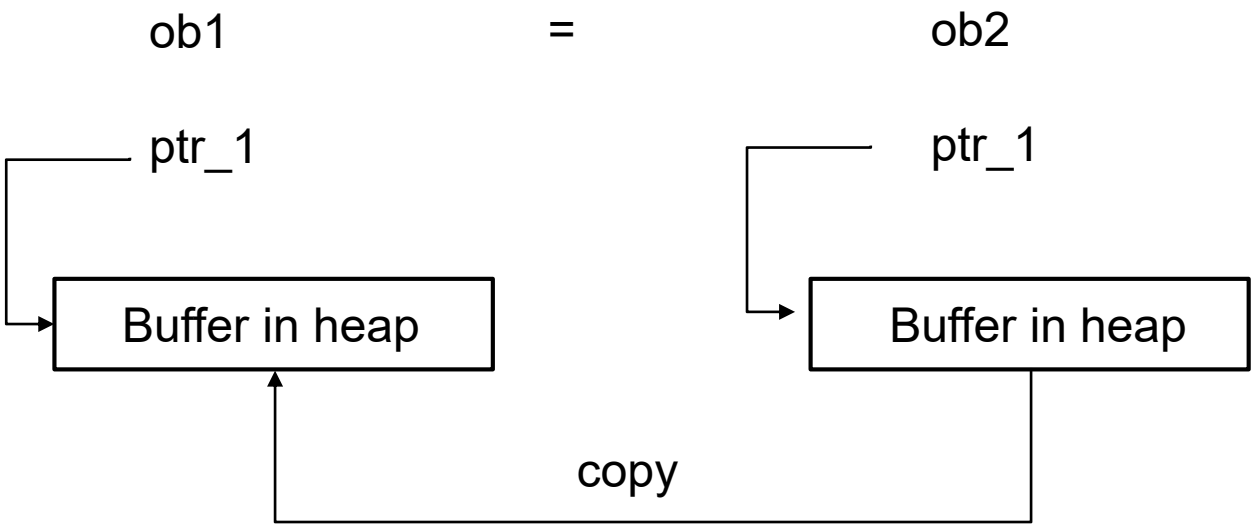


Ten bufor w pamięci heap jest współdzielony przez oba obiekty ob1 i ob2.

➢ **Poprawne rozwiązanie takiego problemu polega na przeciążeniu operatora = .**

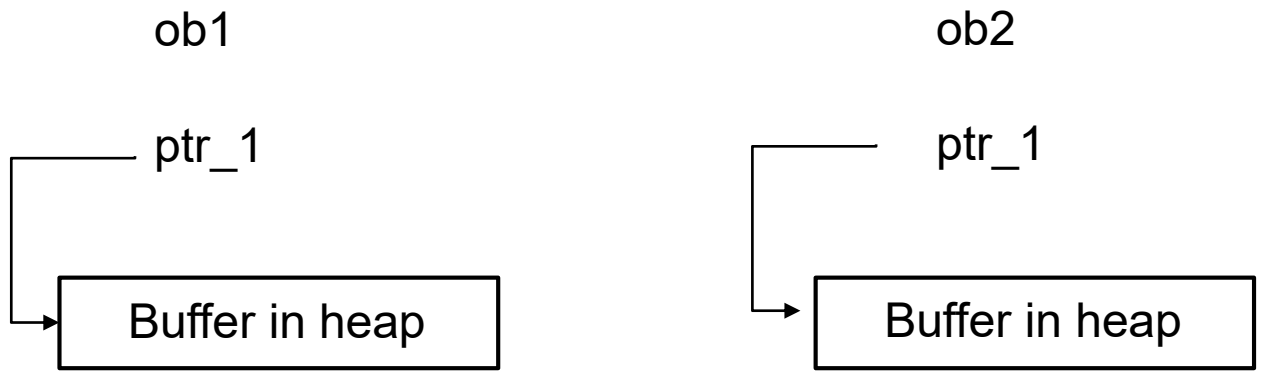


Przeciążenie operatora = (**wersja kopiująca**)

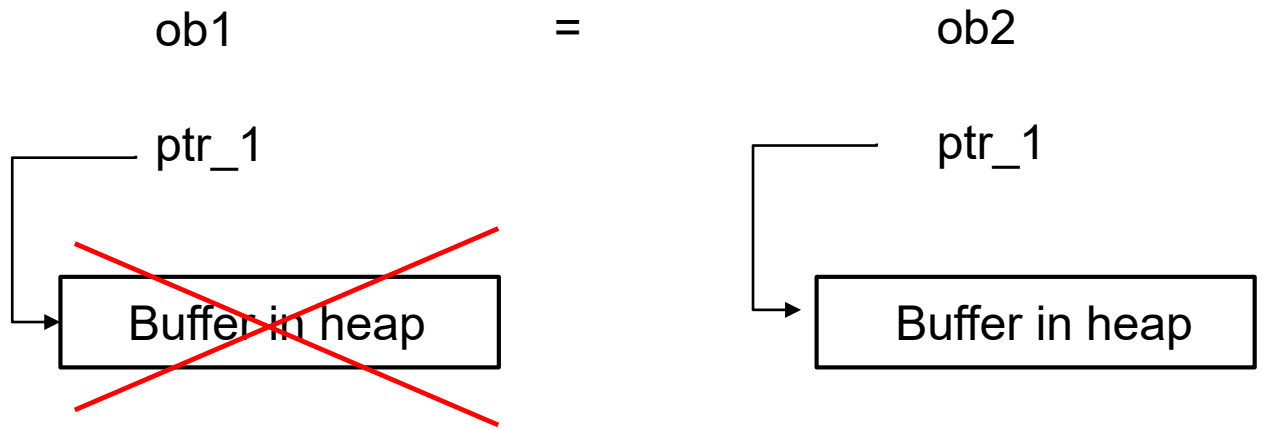


OK

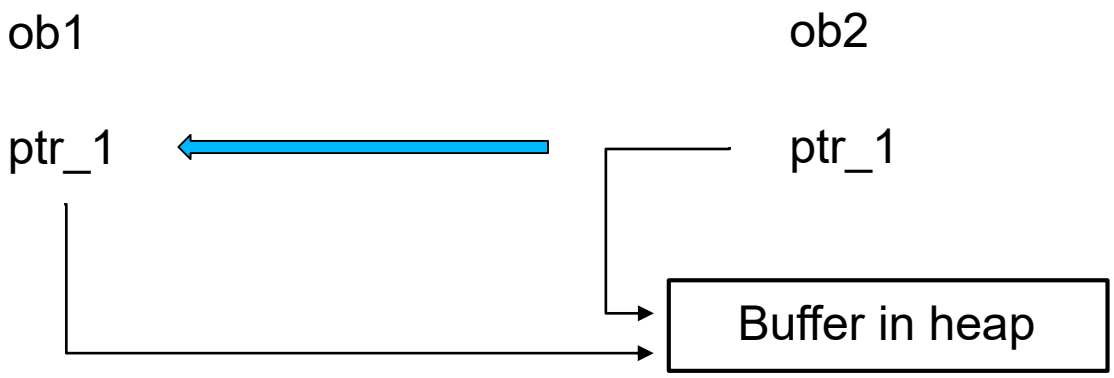
➢ **Poprawne rozwiązanie takiego problemu polega na przeciążeniu operatora = .**



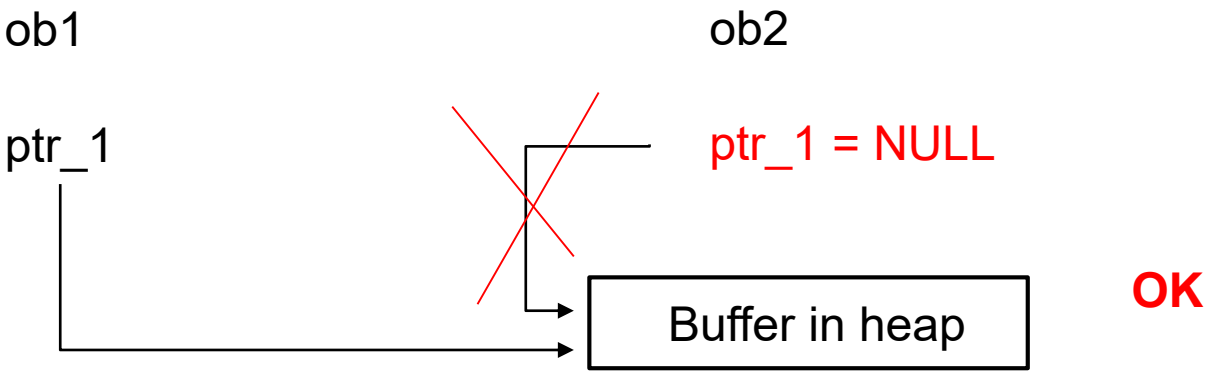
Przeciążenie operatora = (**wersja przenosząca**)



Zwalniamy pamięć dla bufora operandu lewostronnego



Nadpisujemy wskaźnik ptr_1



Operator = przenoszący stosuje się w przypadkach, kiedy operand prawostronny nie wykorzystuje się po zakończeniu przepisania. W takich przypadkach technika przeniesienia podnosi wydajność w porównaniu z techniką kopiowania.

Example W13

Przeciążenie operatorów + - * \

Example W18

```
class coord
{
      double x, y;
public:
      . . . . . .
};
```

```
coord coord::operator + (const coord &right) const
{
      coord ret;
      ret.x = x+right.x;
      ret.y = y+right.y;
      return ret;
}
```

Funkcja-operator + zwraca obiekt typu *coord*. Wewnątrz tej funkcji tworzony jest obiekt pomocniczy *ret*. Można używać tego operatora w wyrażeniach złożonych jako a + b + c, ponieważ a + b + c = (a + b) + c i pierwsza suma musi być przechowywana w pamięci, aby nie zmieniać żadnego z operandów. Dla tego jest używana funkcja, która zwraca obiekt (a nie referencje i nie wskaźnik do obiektu, ponieważ obiekt jest deklarowany lokalnie). Operand prawostronny jest przekazywany przez referencje, aby przyspieszyć wywołanie i uniknąć tworzenia kopii obiektu, a następnie niszczenia tej kopii przez wywołanie destruktora.

```
double coord::operator * ( const coord &right) const
/*=================================================================
dot_prod
=================================================================*/
{
      double ret = 0.0;
      ret = x*right.x+y*right.y;
      return ret;
}
```

Wynikiem iloczynu skalarnego dwóch wektorów jest wartość rzeczywista – funkcja-operator zwraca typ *double*. Pierwszy wektor (operand lewostronny) jest przekazywany przez wskaźnik *this*, a operand prawostronny - jako parametr referencyjny.

```
coord coord::operator * (const double alpa) const
/*=================================================================
mnozenie wektora przez skalar alpa
=================================================================*/
{
      coord ret;
      ret.x = x*alpa;
      ret.y = y*alpa;
      return ret;
}
```

Wynikiem jest wektor (obiekt typu *coord*), a skalar jest przekazywany jako parametr.

Lewy operand musi być obiektem klasy *coord*. Prawy operand musi być skalarem. Odwrotnie (lewy operand - skalar) jest niemożliwe, ponieważ skalar nie jest obiektem klasy *coord*. Operator * generuje wywołanie funkcji-operatora i przekazuje lewy operand przez wskaźnik *this* i prawy operand - jako argument typu *double*.

Przeciążenie operatorów relacyjnych i logicznych

Relacyjne i logiczne operatory zwracają 0 (*false*) or !0 (*true*), dla tego typ wartości zwracanej jest *int* lub *bool*.

Operatory relacyjne i logiczne mogą spotkać się w złożonych wyrażeniach zawierających dane innych typów.

Example W19.

Przeciążenie operatora == jako metody klasy *coord:*

```
int coord::operator == (const coord &right) const
{
        return (x == right.x && y == right.y);
}
```

Operand po lewej stronie jest obiektem klasy *coord*, generuje wywołanie funkcji-operatora i jest przekazywany przez wskaźnik *this*. Operand po prawej stronie jest również obiektem *coord* i jest przekazywany przez referencje. Dane klasy są składowymi wektora, dwa wektory są równe, jeśli każda z odpowiednich składowych są równe pomiędzy sobą.

Przeciążenie operatora > jako metody klasy coord.

```
int coord::operator > (const coord &right) const
{
   double ro_left, ro_right;
   ro_left  = x*x+y*y;
   ro_right = right.x* right.x+ right.y* right.y;
   return (ro_left > ro_ right);
}
```

Dla podanej klasy relacje >, <, >=, <= definiują się na podstawie porównania norm euklidesowych tych wektorów ($|| \ldots ||_2$). Nie oznacza to, że taka definicja jest jednoznaczna. Na przykład, można by było wprowadzić dla porównania normę $|| \ldots ||_\infty$. Zależy to od zagadnienia, które rozwiązujemy.

Operatory-funkcje zaprzyjaźnione do klas

W przypadku funkcji zaprzyjaźnionych wskaźnik *this* nie jest przekazywany - w przypadku operatorów binarnych musimy przekazać zarówno lewy operand, jak i prawy operand.

Pozostałe cechy przeciążania operatorów jako funkcji zaprzyjaźnionych do klas są takie same, jak i dla funkcji-operatorów klasy.

Operatory = oraz [ ] można przeciążyć tylko jako funkcje-metody klasy.

Deklaracja operatora-funkcji zaprzyjaźnionej do klasy (operator binarny):

```
friend typ_ret_val operator # (typ_left  left_operand, typ_right  right_operand);
```

Definicja operatora-funkcji zaprzyjaźnionej do klasy (operator binarny):

```
typ_ret_val operator # (typ_left  left_operand, typ_right  right_operand)
{
   . . . . . .  //body
}
```

Na liście argumentów lewostronny operand jest pierwszym, a prawostronny – drugim, przy tym lewostronny operand może mieć typ odmienny od typu klasy, do której zaprzyjaźniamy nasz operator, ponieważ funkcją zaprzyjaźniona nie dostaje wskaźnika *this* – nie jest metodą klasy.

Przeciążenie operatorów extraction – insertion

```
istream & operator >> (istream & stream, typ_class & ob)
{
        //extractor's body
        return stream;
}
```

Operator >> jest używany w kolejce *mystream* >> *object*, dla tego zwraca referencje do strumienia *mystream*. Operator >> ma dwa operandy – referencje do strumienia stream (lewostronny operand) oraz referencje do obiektu klasy, składowe której odczytujemy ze strumienia *mystream* (prawostronny operand). Jest niemożliwe przeciążyć operator >> jako funkcje-operator składową klasy, ponieważ lewostronny operand ma typ strumienia, a nie klasy.

Example (project_1):

```
class mcoord    //base class
{
protected:
   double x;
   double y;
public:
   mcoord(double xx, double yy) { x = xx; y = yy; }
   mcoord() {x=0; y=0; }
};

class node : public mcoord     //derived class
{
   int numb;
   char str[512];
public:
   node(int nb, char *st, double xx, double yy);
   ~node() { }
……..
   friend ostream & operator << (ostream &stream, const node &ob);
   friend istream & operator >> (istream &stream, node &ob);
   friend ostream & operator << (ostream &stream, const node * ob);
   friend istream & operator >> (istream &stream, node * ob);
};
```

We/Wy sformatowane

```
istream & operator >> (istream &stream, node &ob)
/*==========================================================
Extractor – overload of operator >> for an object of a node class
==========================================================*/
{
   stream >> ob.x >> ob.y >> ob.numb >> ob.str;
   return stream;
}

ostream & operator << (ostream &stream, const node &ob)
/*==========================================================
Inserter – overload of operator << for an object of a node class
==========================================================*/
{
   stream << ob.x << "  " << ob.y << "  " << ob.numb << "  " <<  ob.str << endl;
   return stream;
}
```

We/Wy binarne (niesformatowane)

```
istream & operator >> (istream &stream, my_node *ob)
{
   stream.read((char*)ob, sizeof(*ob));
   if (stream.bad())
   {
         //call error handler
         MY_DATA_GLOBAL::msg.mess(ERR_ACCESS_FILE);
   }
   return stream;
}

ostream & operator << (ostream &stream, const my_node *ob)
{
   stream.write((const char*)ob, sizeof(*ob));
   if (stream.bad())
   {
         //call error handler
         MY_DATA_GLOBAL::msg.mess(ERR_ACCESS_FILE);
   }
   return stream;
}
```

Przeciążenie operatora indeksu [ ].

W C ++ operator [ ] jest operatorem binarnym, i może być przeciążony tylko jako operator-funkcja metoda klasy.

```
return_typ  & class_name :: operator  [ ]  (int index)
{
    //……..
}
```

Zwraca referencje do elementu tablicy o podanym indeksie *index*. Umożliwia to użycie ob [i] =… po lewej stronie operatora przypisania. Operand lewostronny jest obiektem klasy, operand prawostronny jest indeksem elementu tablicy.

Example W20.

Bezpieczna tablica wymaga znacznego nakładu pracy, co powoduje istotny spadek wydajności obliczeń. Z tego powodu w C++ granice tablic domyślnie nie są kontrolowane. Przy tworzeniu bezpiecznej tablicy w tym przykładzie w wersji debug można zrealizować kontrole granic indeksów, jednak w wersji release ominąć tą kontrole.