Laboratorium 6 – przeciążanie operatorów

Przeciążanie operatorów

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

::, .*, ., ?: .

Ten sam operator może być przeciążony więcej niż raz, ale za każdym razem z innym zestawem parametrów: może on być metodą w klasie lub samodzielną funkcją. Natomiast nie można utworzyć jednocześnie metody i funkcji z tymi samymi parametrami.
Należy zdefiniować funkcję o nagłówku:

operator nazwa_operatora(lista_parametrów)

Jeżeli funkcja przeciążająca dany operator występuje jako metoda, to lewy argument operacji musi być obiektem danej klasy (jest on przekazywany przez wskaźnik this).

Cztery operatory: =, [ ], (), -> muszą być definiowane jako metody.

Operator przypisania = nie może być dziedziczony i w klasie pochodnej musi być skonstruowany oddzielnie.

```
class L_zesp{
private:
        int re, im;
public:
        //inne metody zadeklarowane  w klasie
        L_zesp(int a=0,b=0)
        {re=a; im=b;}
        void operator +(L_zesp &B)
        {// wynik działania jest w obiekcie, na którym działamy aktualnie!!!
        re=re+B.re;
        im=im+B.im;
        }
        void druk()
        {//drukuje liczbę zespoloną;
        }
}
```

Wywołanie w programie głównym np.:

Klasa A1(2,6), A2(7,-3), A3;

A1.druk(); //z=[2+i*6]

A1+A2*;* //tutaj wywołany został ten przeciążony operator +

A1.druk(); //z=[-5+i*3]

Drugą możliwością jest sytuacja, gdy metoda, która przeciążą operator zwraca wynik typu *L_Zesp*, a nie *void* (sposób bardziej naturalny, bo bliższy notacji matematycznej):

```
L_zesp operator +(L_zesp & B)

        {       L_zesp w; //wynik dodawania będzie w nowej zmiennej
```

```
        w.re=re+B.re;

        w.im=im+B.im;

        return w;

    }
```

wtedy:

L_zesp A1(2,6), A2(7,-3), A3;

A1.druk(); //z=2+i*6

A3.druk(); //z=0

A3=A1+A2; //tutaj wywołany został ten przeciążony operator +

A3.druk(); //z=-5+i*3

Przeciążenie operatora + nie przeciąża automatycznie operatora +=. Trzeba to zrobić w oddzielnej metodzie. Analogicznie dla innych tego typu operatorów.

Sprawdzanie warunków – funkcja assert()

Funkcja assert z biblioteki assert.h służy do debuggowania programów:

void assert (int expression);

Gdy testowany w niej warunek logiczny - *expression* przyjmuje wartość fałsz, na standardowe wyjście błędów wypisywany jest komunikat o błędzie (zawierające m.in. argument wywołania makra; nazwę funkcji, w której zostało wywołane; nazwę pliku źródłowego oraz numer linii w formacie zależnym od implementacji) i program jest przerywany poprzez wywołanie funkcji abort.

Jeżeli przez dodaniem biblioteki dodamy *#define NDEBUG*, wtedy wyłączone zostanie assert.

```
#include <iostream>
#include <assert.h>
using namespace std;
int main ()
{  int a=10, * b = NULL, * c = NULL;
   b=&a;
  assert (b!=NULL);
  cout<<*b<<"\n";
  assert (c!=NULL);
  cout<<*c<<"\n";
  return 0;
}
```

Typy wyliczeniowe

W kodzie często wygodniej jest posługiwać się nazwami stałych zamiast wartościami liczbowymi. Dlatego często wewnątrz klas definiuje się typy wyliczeniowe.

```
class Zakupy
{
public:
        enum Ilosc {MALO,SREDNIO,DUZO};
        Ilosc mZakupy;                  //wykorzystanie typu wewnątrz klasy
        Zakupy()
        {
                mZakupy=MALO;       //wykorzystanie wartości wewnątrz klasy
        }
};
…
        //gdzieś poza klasą
        Zakupy::Ilosc jakieZakupy;
        jakieZakupy=Zakupy::SREDNIO;
…
```

Warto też wiedzieć, że kolejnym wartościom typu wyliczeniowego odpowiadają kolejne wartości całkowite począwszy od 0. Tak więc w powyższym przykładzie wartości MALO odpowiada 0, wartości SREDNIO odpowiada 1 a wartości DUZO odpowiada 2. Przypisanie wartości całkowitej do zmiennej typu wyliczeniowego może wymagać jawnego rzutowania.

Zadanie 8

Do klasy pies:

- dodać pole płeć. Pole ma wykorzystywać typ wyliczeniowy.
- zmodyfikować metodę naLancuch() tak aby wyświetlała płeć.
- przeciążyć operator +. Operator za pomocą metody assert() z biblioteki assert.h ma sprawdzić płeć obu psów i jeżeli „działanie” ;-) jest wykonalne, to zwraca nowy obiekt. Płeć nowego psa ma być losowa. Imię nowego psa ma być kombinacją imion dodawanych psów (można wykorzystać generator liczb pseudolosowych oraz metodę substr() klasy string)

Obiekty funkcyjne – operator()

Celem wprowadzenia obiektów funkcyjnych jest zastąpienie wskaźników na funkcje. Do stworzenia funktora (obiektu funkcyjnego) wykorzystuje się przeciążony operator():

```
struct nazwa
{

        //operator() może też zwracać wartość określonego typu (nie musi być void)
        virtual void operator()() const
        { //instrukcje ...}
};
```

Obiekt funkcyjny jest już zadeklarowany. Aby z niego skorzystać, należy stworzyć jego instancję, z której można korzystać jak ze zwykłej funkcji bądź tez wskaźnika na nią:

nazwa f; //instancja obiektu funkcyjnego

f(); // użycie operatora ()

Taki obiekt funkcyjny może zostać przekazany do innej funkcji.

```
void inna_funkcja(const nazwa &f)  //definicja funkcji, która jako parametr
                                   //ma przekazany taki obiekt
{
        f(); //wywołanie
}

/
inna_funkcja(f);   // przekazanie istniejącego obiektu
```

Zadanie 9

Stworzyć klasę Miska. W klasie Miska:

- przeciążyć operator(), który będzie miał jeden argument typu int oznaczający porcję jedzenia.
- zdefiniować również pole prywatne przechowujące wartość porcji
- zdefiniować publiczną metodę dostępową dajPorcję() która zwraca ilość jedzenia

W klasie Pies

- przeciążyć operator <<, tak żeby można było nakarmić psa przekazując mu obiekt klasy Miska. Głód psa powinien się zmniejszyć natomiast miska powinna zostać opróżniona

Klasy i funkcje zaprzyjaźnione

Czasami chce się, aby pewne funkcje nie będące metodami klasy miały dostęp do składowych niepublicznych tej klasy. Efekt ten można osiągnąć definiując metody publiczne klasy udostępniające dane prywatne zawarte w obiekcie. Wtedy jednak każda funkcja może ich użyć i uzyskać dostęp do tych danych. Kiedy chce się je udostępnić tylko wybranym funkcjom, to muszą to być funkcje „zaprzyjaźnionym" z tą klasą.

Funkcja taka musi być zadeklarowana wewnątrz klasy, z którą jest zaprzyjaźniona. Robi się to dodając na początku jej deklaracji modyfikator **friend**.

```
class Klasa {
        // ...
        friend typ_funkcji nazwafunkcji(lista_parametrów);
        // ...
    };
```

Deklarację przyjaźni można umieścić w dowolnej sekcji definicji klasy, publicznej, prywatnej lub chronionej - nie ma to żadnego znaczenia. Funkcja zaprzyjaźniona z klasą **nie jest** metodą tej klasy. Funkcja taka ma dostęp do wszystkich składowych klasy. Nie ma ona jednak określonego wskaźnika **this** (bo nie jest składową klasy) i nie jest wywoływana na rzecz obiektu. Jedna funkcja

może być zaprzyjaźniona z wieloma klasami: deklaracja przyjaźni musi być wtedy zawarta w definicji każdej z tych klas. Ciało takiej funkcji umieszcza się poza klasą i już bez słowa friend:

```
typ_funkcji nazwafunkcji(lista_parametrów)
{ //instrukcje funkcji
}
```

Przykładu poprzedniego c.d. Klasa L_zesp deklaruje przyjaźń z funkcją przeciążającą operator -:

```
friend L_zesp operator -(L_zesp &A,L_zesp &B);
```

Natomiast definicja tej funkcji jest poza klasą. Ma ona postać:

```
L_zesp operator -(L_zesp &A,L_zesp &B)
{       L_zesp R;
        R.re=A.re-B.re;
        R.im=A.im-B.im;
        return R;
};
```

Można zadeklarować w klasie przyjaźń ze wszystkimi metodami innej klasy:

```
class B; //deklaracja zapowiadająca klasę B
class A {
   // ...
   friend class B;
   //...
};
```

Przyjaźń **nie jest** relacją:

- zwrotną - jeśli klasa **B** jest zaprzyjaźniona z klasą **A**, to nie oznacza to wcale, że klasa **A** jest zaprzyjaźniona z klasą **B**. Trzeba to jawnie zadeklarować;
- przechodnią - jeśli klasa **A** jest zaprzyjaźniona z klasą **B** i klasa **B** jest zaprzyjaźniona z klasą **C**, to nie oznacza to wcale, że klasa **A** jest zaprzyjaźniona z klasą **C**;
- dziedziczną - klasy potomne nie dziedziczą przyjaźni od swoich klas bazowych.

Operator << do wyświetlania definiuje się zazwyczaj jako funkcję zaprzyjaźnioną. Przykład przeciążania operatora << ze strumieniami:
W klasie L_zesp należy zadeklarować przyjaźń:

```
friend ostream & operator<< (ostream & ob, L_zesp & liczba);
```

Natomiast poza klasą zdefiniować funkcję:

```
ostream & operator<< (ostream & ob, L_zesp & liczba)
{
        ob << "Czesc rzeczywista: " << liczba.re << endl;
        ob << "Czesc urojona: " << liczba.im << endl;
        return ob;
}
```

Użycie tego operatora ma wtedy postać: *cout<<A3;*

Zadanie 10

Dla klasy Zwierze zdefiniować (jako funkcję zaprzyjaźnioną) operator <<, aby wyświetlał stan zwierzęcia (zamiast metody naLancuch()).

Zadanie 11

Stworzyć klasę Czlowiek. W klasie Czlowiek:

- zdefiniować pola prywatne: imie, plec i wiek. Użyć typu wyliczeniowego.
- konstruktor inicjujący pola.
- metodę wyświetlającą dane człowieka.
- publiczną metodę karmPsa(), jej parametrem jest pies do nakarmienia oraz ilość pożywienia. Metoda powinna korzystać z metody jedz() psa, opisanej poniżej.

Tylko człowiek może karmić swojego najlepszego przyjaciela, tak należy więc w klasie Pies:

- zdefiniować prywatną metodę jedz(), której argumentem jest obiekt klasy Miska
- zdefiniować przyjaźń z klasą Czlowiek.

Sprawdzić działanie zdefiniowanych metod