C++ - funkcje zaprzyjaźnione

Dostęp do składowych prywatnych

- **W języku C++ jednostką chronioną przed niepowołanym dostępem jest klasa a nie obiekt. Oznacza to, że funkcja składowa danej klasy może używać wszystkich składowych prywatnych dowolnego obiektu tej samej klasy**
- **Do składowych prywatnych jakiegoś obiektu nie ma dostępu funkcja innej klasy, ani jakakolwiek funkcja niezależna.**
- **Deklaracja zaprzyjaźnienia pozwala zadeklarować w danej klasie funkcje, które mogą wykorzystywać jej składowe prywatne – dane i funkcje.**

Funkcja zewnętrzna zaprzyjaźniona z klasą

- **Funkcji niezależnej zostają udostępnione składniki prywatne klasy A, np.:**

```
class A
{     ...
      //deklaracja zaprzyjaźnienia z funkcją niezależną fun
      friend  typ fun(argumenty);
      ...
};
// definicja niezależnej funkcji zaprzyjaźnionej
```

- **Zwykle funkcja zaprzyjaźniona ma argument lub zwraca wartość typu A, co uzasadnia deklarację zaprzyjaźnienia.**

Zadanie 1

- **Zdefiniować klasę macierz2x2:**

  –o polu prywatnym będącym tablicą 2 x 2

  –z konstruktorem, którego argumentem ma być tablica 2 x 2

- **Napisać niezależną funkcję det, obliczającą wyznacznik macierzy 2 x 2 (obiektu klasy macierz2x2)**

-

```
//klasa macierz2x2
//z niezalezna funkcja zaprzyjazniona det
class macierz2x2
{    float a[2][2];
     public:
          macierz2x2(float b[2][2]);
          friend float det(macierz2x2);   //deklaracja
                                   //zaprzyjaźnienia
};
macierz2x2::macierz2x2(float b[2][2])
{ for (int i=0;i<2;i++)  for (int j=0;j<2;j++) a[i][j]=b[i][j]; }
float det(macierz2x2 c) //funkcja zaprzyjazniona
{  return c.a[0][0]*c.a[1][1]-c.a[0][1]*c.a[1][0];}
main()
{
     float q[][2]={{1,2},{4,5}};
     macierz2x2 m(q);
     float d=det(m);
     if (d)
          cout<<"\nMacierz nieosobliwa ";
     else
          cout<<"\nMacierz osobliwa";
     cout<<"det(m)="<<d;
}
```

Zadanie 2

- **Zdefiniować klasę student posiadającą:**

  –pola prywatne: nazwisko, ile_ocen, oceny (tablica ocen tworzona dynamicznie)

  –konstruktor o dwóch argumentach domniemanych, który ma ponadto utworzyć tablicę dynamiczną *oceny* i zainicjalizować pola tej tablicy zerami

  –konstruktor o trzech argumentach wejściowych, który ma utworzyć tablicę dynamiczną *oceny* i zainicjalizować odpowiednie pola wartościami argumentów

  –destruktor zwalniający pamięć przydzieloną na tablicę ocen

  –funkcję wyświetlającą nazwisko i listę ocen danego obiektu *student*

  –napisać niezależną funkcję obliczającą średnią ocen dla obiektu klasy *student*

```
class student {
     char nazwisko[80];
     int ile_ocen;
     float *oceny;  //tablica ocen tworzona dynamicznie
  public:
     friend float srednia(student &);
     student(char *napis="", int ile=0);
     student(char *,float [],int);
     ~student(){delete oceny;}
     void wyswietl();
};   //koniec klasy
```

Konstruktory klasy

```
student::student(char *napis, int ile) //tu bez przypisania
{    strcpy(nazwisko,napis);
     ile_ocen=ile;
     oceny=new float[ile_ocen];
     for (int i=0;i<ile_ocen;i++) *(oceny+i)=0;
}
student::student(char *napis,float t[],int ile)
{    ile_ocen=ile;
     oceny=new float[ile_ocen];
     strcpy(nazwisko,napis);
     for (int i=0;i<ile_ocen;i++) *(oceny+i)=t[i];
}
```

Przykład 2 – inne metody klasy

```
void student::wyswietl()
{    cout<<"\nStudent: "<<nazwisko<<", oceny: ";
     if (ile_ocen)
       for(int i=0;i<ile_ocen;i++) cout<<*(oceny+i)<<" ";
     else cout<<"brak danych ";
}
float srednia(student &s)
{    float suma=0;
     if (s.ile_ocen)
     {    for(int i=0;i<s.ile_ocen;i++) suma+=*(s.oceny+i);
     suma/=s.ile_ocen;        }
  return suma;
}
void main()
{    float a[]={3,4.5,3.5},b[]={2,3,4};
     student s1;
     student s2("Kowalski Jan", a, sizeof(a)/sizeof(a[0]));
     student s3("Nowak Anna",b,3);
     s1.wyswietl(); cout<<" srednia: "<<srednia(s1);
     s2.wyswietl(); cout<<" srednia: "<<srednia(s2);
     s3.wyswietl();cout<<" srednia: "<<srednia(s3);
     s1=s2;
     s1.wyswietl();
}
```

Funkcja klasy zaprzyjaźniona z funkcją innej klasy

- **Funkcji składowej klasy B zostają udostępnione składowe prywatne klasy A, np.:**

```
class B;  //musi się pojawić jeżeli definicja klasy B
               //nie jest umieszczona wcześniej
class A
{    ...
     //deklaracja zaprzyjaźnienia z funkcją klasy B
     friend  typ B::fun(argumenty);
};
class B
{    //ciało klasy B
}
```

Funkcja klasy zaprzyjaźniona z funkcją innej klasy

- **Zwykle taka deklaracja zaprzyjaźnienia spowodowana jest tym, że funkcja składowa fun ma argument lub rezultat typu A**
- **Aby przetłumaczyć jej deklarację (wewnątrz deklaracji A) kompilator musi wiedzieć, że A jest klasą**
- **Natomiast aby przetłumaczyć definicję fun, kompilator musi dysponować kompletną deklaracją A.**

Zaprzyjaźnienie całej klasy

- **Zamiast deklarować zaprzyjaźnienie kolejno dla wszystkich funkcji składowych B, używa się w deklaracji klasy A globalnej deklaracji zaprzyjaźnienia:**

```
class B;  // musi się pojawić jeżeli definicja klasy B
               // nie jest umieszczona wcześniej
class A
{    ...
     //deklaracja zaprzyjaźnienia z wszystkimi funkcjami klasy B
     friend  class B;
};
class B
{    //ciało klasy B
}
```

Przykład 3

- **Opracować dwie klasy:**

  – **wekt umożliwiającą obsługę wektorów o N składowych (początkowo przyjąć N=3). Konstruktor opracować tak, aby istniała możliwość deklarowania wektora z jednoczesnym podaniem wartości wszystkich trzech składowych albo inicjalizacja samymi zerami. Dodać funkcję wyświetlającą składowe wektora w postaci <x1,x2,...,xn>.**

  – ***macierz*** **do reprezentowania macierzy *N x N.***
  Argumentem konstruktora ma być tablica *N x N* zawierająca wartości do inicjalizacji macierzy.

  – **W klasie wekt zdefiniować funkcję *iloczyn (zaprzyjaźnioną z klasą macierz)* obliczającą wektor równy iloczynowi macierzy przez wektor oraz program testowy.**

Klasa macierz

```
// klasa wekt z funkcja zaprzyjazniona
// z klasa macierz:
const int N=3;
class wekt; //niezbedna do kompilacji klasy macierz
class macierz
{ float a[N][N];
  public:
  macierz(float b[N][N]);
  wekt iloczyn(const wekt &v); // f. zaprzyjazniona
};
class wekt
{ float x[N];
  public:
     wekt(float c[N]);
     wekt();
     void wyswietl();
     // deklaracja zaprzyjaznienia:
     friend wekt macierz::iloczyn(const wekt &v);
     //friend class macierz; //zaprzyjaznienie
     //z wszystkimi funkcjami klasy macierz
};
void wekt::wyswietl()
{    cout<<"<";
     for (int i=0;i<N;i++) cout<<x[i]<<" ";
     cout<<">\n";
}
```

```
wekt::wekt()
{ for (int i=0;i<N;i++) x[i]=0; }
wekt::wekt(float c[N])
{ for (int i=0;i<N;i++) x[i]=c[i]; }
macierz::macierz(float b[N][N])
{ for (int i=0;i<N;i++)
     for (int j=0;j<N;j++) a[i][j]=b[i][j]; }
```

Definicja funkcji z klasy wektor zaprzyjaźnionej z klasą macierz

```
wekt macierz::iloczyn(const wekt &v)
{ wekt w;
  float p=0;
  for (int i=0;i<N;i++)
  {   p=0;
       for (int j=0;j<N;j++) p+=a[i][j]*v.x[j];
          w.x[i]=p;
  }
  return w;
 }
void main()
{ float w[]={1,1,1},w1[]={0,0,0};
  wekt v1,v=wekt(w);
  cout<<"v=";v.wyswietl();
  float q[][N]={{1,2,3},{4,5,6},{7,8,9}};
  macierz m(q);
  v1=m.iloczyn(v); cout<<"m.v=";v1.wyswietl();
  v=wekt(w1);
  v1=m.iloczyn(v); cout<<"m.v=";v1.wyswietl();
  //v1=v.iloczyn(m);  blad!!!
}
```

Przykład 3 – funkcja niezależna

- **Można zdefiniować funkcję niezależną postaci:**

```
wekt iloczyn(const wekt &a, const macierz &b);
```

- **Taka funkcja musi być wówczas zaprzyjaźniona zarówno z klasą wekt jak i klasą macierz**

Przykład 4 – wersja na wskaźnikach

```
class wekt;
class macierz
{ float *a;
```

```
  public:
     const static int N;
     macierz(float *b);
      ~macierz(){ delete [ ] a;}
     wekt iloczyn(const wekt &v); //f. zaprzyjazniona
};
class wekt  //deklaracja klasy
{ float *x;
  public:
     const static int N;
     wekt(float *c);
     wekt();
     ~wekt() { delete [ ] x; }
     void wyswietl();
     friend wekt macierz::iloczyn(const wekt &v);     //friend
};
void wekt::wyswietl()
{    cout<<"<";
     for (int i=0;i<N;i++) cout<< *(x+i) <<" ";
     cout<<">\n";  }
wekt::wekt()
{  x=new float[N]; for (int i=0;i<N;i++) *(x+i)=0; }
wekt::wekt(float *c)
{ x=new float[N]; for (int i=0;i<N;i++) *(x+i)=*(c+i); }
macierz::macierz(float *b)
{ a=new float[N*N];
  for (int i=0;i<N*N;i++) *(a+i)=*(b+i);  }
wekt macierz::iloczyn(const wekt &v)
{ wekt w;
  float p=0;
  for (int i=0;i<N;i++)
  {   p=0;
       for (int j=0;j<N;j++) p+=*(a+N*i+j)*(*(v.x+j);
      *(w.x+i)=p;
  }
  return w;
 }
const int wekt::N=4;
const int macierz::N=4;


void main()
{ int ile=wekt::N;  int ile2=pow(ile,2);  float *w,*w1;
  w=new float[ile];   w1=new float[ile];
  for (int i=0;i<ile;i++) *(w+i)=1;    //wektor pomocniczy 1
  for (i=0;i<ile;i++) *(w1+i)=0;   //wektor pomocniczy 2
  wekt v1,v=wekt(w);          //deklaracja i inicjalizacja obiektu
  cout<<"v=";v.wyswietl();
  float *q;  q=new float[ile2];
```

```
  for (i=0;i<ile2;i++) *(q+i)=i+1; //macierz pomocnicza
  macierz m(q); v1=m.iloczyn(v);
  cout<<"m.v=";v1.wyswietl();
  v=wekt(w1);
  v1=m.iloczyn(v);
  cout<<"m.v=";v1.wyswietl();
  delete []q,[]w,[]w1;
}
```