Język C++ zajęcia nr 12

1. Funkcje zaprzyjaźnione

Funkcja, która nie jest składową pewnej klasy, nie ma dostępu do jej składowych prywatnych. **Funkcja zaprzyjaźniona** z pewną klasą jest funkcją, która ***nie będąc funkcją składową*** tej klasy ma dostęp do jej wszystkich składowych (również prywatnych i zabezpieczonych). Dostęp ten uzyskiwany jest w standardowy sposób poprzez operatory składowej **.** lub **->** . Stosowanie funkcji zaprzyjaźnionych pozwala wewnątrz pewnej funkcji na równoczesny dostęp do składowych prywatnych dwóch różnych klas.

Zadeklarowanie przyjaźni funkcji z pewną klasą następuje poprzez umieszczenie **w tej klasie** *deklaracji* wybranej funkcji poprzedzonej słowem

`friend`

Można zadeklarować przyjaźń całej klasy P (wszystkich jej funkcji składowych) z inną klasą R poprzez umieszczenie w klasie R deklaracji:

`friend class P`

Wprowadź poniższy program, ZINTERPRETUJ POSZCZEGÓLNE ELEMENTY KODU ŹRÓDŁOWEGO!!! Skompiluj i uruchom program.

Funkcja zaprzyjaźniona z dwiema klasami:

```
#include <iostream>
using namespace std;
class romb;
class kwadrat
{
   float a;     // długość boku kwadratu
   float pole;
 public:
   kwadrat(float x);
   friend float suma_pol(kwadrat&,romb&);     // deklaracja przyjaźni
};
```

```
class romb
{
   float a,b;   // długości przekatnych rombu
   float pole;
 public:
   romb(float x, float y);
   friend float suma_pol(kwadrat&,romb&);     // deklaracja przyjaźni
};

kwadrat::kwadrat(float x)
{
   a=x;
   pole=a*a;
}

romb::romb(float x,float y)
{
   a=x;
   b=y;
   pole=a*b/2;
}

float suma_pol(kwadrat& k,romb& r)
{
   //------------------- dostęp do prywatnych danych składowych obydwu klas
   return k.pole+r.pole;
}

int main()
{
   kwadrat kwd(7.5);
   romb rmb(7,5);
   cout<<suma_pol(kwd,rmb);
}
```

Oczekiwane wyniki:

```
73.75
```

Przeciążenie operatora strumieniowego << znacznie upraszcza składnię operacji wyprowadzania wartości obiektów klas do strumienia wyjściowego. Funkcja operatorowa **operator<<** nie może być składową klasy obiektu, który podlega operacji wyprowadzania (lewy argument operatora **<<** jest obiektem strumieniowym). Funkcja ta nie może być również zwykłą funkcją globalną w przypadku konieczności dostępu do składowych prywatnych wyprowadzanego obiektu. Rozwiązanie problemu wymaga zdefiniowania **funkcji operatorowej zaprzyjaźnionej** z daną klasą.

ZINTERPRETUJ POSZCZEGÓLNE ELEMENTY KODU ŹRÓDŁOWEGO!!!

Funkcje zaprzyjaźnione i przeciążenie operatorów:

```
#include <iostream>
using namespace std;

class complex
{
  private:
    float re_part;
    float im_part;
  public:
  //-------------------- Konstruktory
    complex(){re_part=0; im_part=0;}
    complex(float re, float im=0) {re_part=re; im_part=im;}
  //-------------------- Deklaracje funkcji operatorowych
    complex operator+ (complex& arg2);
    complex operator- (complex& arg2);
    int operator< (complex& arg2);
    int operator> (complex& arg2);
  //-------------------- Deklaracja funkcji zaprzyjaźnionej
    friend ostream& operator<< (ostream& out,complex& c);
};
```

```
//-------------------- Definicje funkcji operatorowych
complex complex::operator+ (complex& arg2)
{
  complex sum;
  sum.re_part = re_part + arg2.re_part;
  sum.im_part = im_part + arg2.im_part;
  return sum;
}
complex complex::operator- (complex& arg2)
{
  complex dif;
  dif.re_part = re_part - arg2.re_part;
  dif.im_part = im_part - arg2.im_part;
  return dif;
}
int complex::operator< (complex& arg2)
{
  float md1, md2;
  md1 = re_part*re_part + im_part*im_part;
  md2 = arg2.re_part*arg2.re_part + arg2.im_part*arg2.im_part;
  return md1 < md2;
}
int complex::operator> (complex& arg2)
{
  float md1, md2;
  md1 = re_part*re_part + im_part*im_part;
  md2 = arg2.re_part*arg2.re_part + arg2.im_part*arg2.im_part;
  return md1 > md2;
}
```

```
//-------------------- Definicja funkcji zaprzyjaźnionej
ostream& operator<< (ostream& out,complex& c)
{
    out<<"("<<c.re_part<<","<<c.im_part<<")";
    return out;
}

int main()
{
    complex a;
    complex b=5;
    complex c(1,2);
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
    complex d = b+c+b;
    cout << "d = " << d << endl;
    complex e = b-c;
    cout << "e = " << e << endl;
    if(b<c) cout << "liczba b jest mniejsza od c" << endl;
    else cout << "liczba b nie jest mniejsza od c" << endl;
}
```

Oczekiwane wyniki:

```
a = (0,0)
b = (5,0)
c = (1,2)
d = (11,2)
e = (4,-2)
liczba b nie jest mniejsza od c
```

2. Podobiekty

Składowe pewnej klasy X mogą być obiektami innej klasy Y. Utworzone obiekty klasy X zawierają **podobiekty** klasy Y. Konstruowanie (inicjalizowanie) podobiektów wykonywane jest przez odpowiednie konstruktory ich klasy.

Kolejność wykonywania konstruktorów:

Najpierw wykonywane są konstruktory podobiektów w kolejności deklarowania składowych klasy, a następnie wykonywany jest konstruktor obiektu zawierający te podobiekty.

Kolejność wykonywania destruktorów:

Najpierw wykonywany jest destruktor obiektu zawierającego podobiekty, a następnie wykonywane są destruktory tych podobiektów w odwrotnej kolejności do ich konstruktorów.

3. Tablice obiektów

Problem podobny do konstruowania podobiektów występuje przy definiowaniu tablicy, której elementami są obiekty pewnej klasy. W procesie tworzenia takiej tablicy, kolejno dla każdego jej elementu zostaje wywołany odpowiedni konstruktor. W przypadku, gdy definiowana tablica jest równocześnie inicjalizowana, dla inicjalizowanych elementów wywoływany jest konstruktor zgodny z postacią odpowiedniego wyrażenia na liście inicjalizatora, a dla pozostałych elementów wywoływany jest konstruktor domniemany klasy. Konstruktor domniemany wywoływany jest również, gdy tablica nie jest inicjalizowana. Przy likwidowaniu tablicy każdy jej element oddzielnie podlega działaniu destruktora klasy, przy czym destruktory poszczególnych elementów tablicy są wywoływane w kolejności odwrotnej („od końca”).

Wprowadź poniższy program, ZINTERPRETUJ POSZCZEGÓLNE ELEMENTY KODU ŹRÓDŁOWEGO!!! Skompiluj i uruchom program.

Tablica obiektów:

```
#include <iostream>
using namespace std;

class zwierzak
{
  public:
    char* nazwa;
    zwierzak(char* n);
    zwierzak();
    ~zwierzak();
};

ostream& operator<<(ostream& out,zwierzak& zw)
{
    return out<<zw.nazwa;
}

zwierzak::zwierzak(char* n)
{
    cout<<"konstruktor JEDNOARGUMENTOWY: "<<n<<endl;
    nazwa=n;
}

zwierzak::zwierzak()
{
    cout<<"konstruktor DOMNIEMANY"<<endl;
    nazwa="<---->";
}

zwierzak::~zwierzak()
{
    cout<<"destruktor: " << nazwa << endl;
}
```

```
int main()
{
  zwierzak klatka[3];
  zwierzak zoo[7]={"Tygrys","Lew","Krokodyl","Sowa","Zebra"};
}
```

```
konstruktor DOMNIEMANY
konstruktor DOMNIEMANY
konstruktor DOMNIEMANY
konstruktor JEDNOARGUMENTOWY: Tygrys
konstruktor JEDNOARGUMENTOWY: Lew
konstruktor JEDNOARGUMENTOWY: Krokodyl
konstruktor JEDNOARGUMENTOWY: Sowa
konstruktor JEDNOARGUMENTOWY: Zebra
konstruktor DOMNIEMANY
konstruktor DOMNIEMANY
destruktor: <---->
destruktor: <---->
destruktor: Zebra
destruktor: Sowa
destruktor: Krokodyl
destruktor: Lew
destruktor: Tygrys
destruktor: <---->
destruktor: <---->
destruktor: <---->
```

4. Dziedziczenie

4.1. Klasa bazowa i klasa pochodna

Język C++ pozwala zdefiniować nową klasę metodą dziedziczenia własności pewnej innej klasy. Poprzez dziedziczenie powstaje nowa klasa – **klasa pochodna**, która automatycznie zawiera składowe (dane i funkcje) określone w zdefiniowanej już **klasie bazowej**.

W klasie pochodnej (oprócz odziedziczonych automatycznie składowych klasy bazowej) można dodatkowo zdefiniować:

- Dodatkowe dane składowe o identyfikatorach różnych od istniejących w klasie bazowej.
- Dodatkowe funkcje składowe o identyfikatorach różnych od istniejących w klasie bazowej.
- Dane składowe o identyfikatorach identycznych z istniejącymi w klasie bazowej (*efekt zasłaniania*).
- Funkcje składowe o identyfikatorach identycznych z istniejącymi w klasie bazowej (*efekt zasłaniania*).

Klasa pochodna może powstać poprzez dziedziczenie **jednobazowe** (jedna klasa bazowa) lub **wielobazowe** (wiele klas bazowych). Każda klasa bazowa sama może być pochodną innej klasy bazowej tworząc dziedziczenie **sekwencyjne**.

Definicja klasy pochodnej dziedziczącej jednobazowo:

```
class  nazwa_klasy_pochodnej : prawo_dostępu  nazwa_klasy_bazowej
{
   // ... ciało klasy ...
};
```

Na **liście dziedziczenia** jako **prawo dostępu** umieszczane jest jedno ze słów kluczowych: **`public`**, **`private`**, **`protected`**.

Jeżeli prawo dostępu nie jest jawnie podane, przyjmuje się domyślnie prawo dostępu **`private`**. Prawo dostępu określa sposób, w jaki składowe klasy bazowej dostępne są w klasie pochodnej.

- Prawo dostępu **`public`** powoduje, że składowe publiczne klasy bazowej pozostają publicznymi składowymi klasy pochodnej, a składowe zabezpieczone klasy bazowej pozostają zabezpieczonymi składowymi klasy pochodnej.
- Prawo dostępu **`protected`** powoduje, że składowe publiczne i zabezpieczone klasy bazowej stają się zabezpieczonymi składowymi klasy pochodnej.
- Prawo dostępu **`private`** powoduje, że składowe publiczne i zabezpieczone klasy bazowej stają się prywatnymi składowymi klasy pochodnej.

4.2. Dostęp do składników klasy bazowej

Wszystkie składowe klasy bazowej (zarówno **`public`**, jak i **`private`** oraz **`protected`**) są dziedziczone przez klasę pochodną, co oznacza, że wchodzą w skład komponentów tej klasy. Nie wszystkie jednak składowe klasy bazowej są dostępne w klasie pochodnej.

Reguły dostępu:

- **Składowe prywatne (`private`) klasy bazowej nie są dostępne** w klasie pochodnej z wyjątkiem funkcji klasy pochodnej **zaprzyjaźnionych** z klasą bazową.
- **Składowe nieprywatne (`public`, `protected`) klasy bazowej są dostępne** w klasie pochodnej. Składowa zabezpieczona (**`protected`**) klasy bazowej **nie jest dostępna** poza klasą pochodną.

Dziedziczenie umożliwia wywoływanie funkcji składowych pewnej klasy nie tylko dla własnych obiektów, ale także dla obiektów klas pochodnych.

Funkcja klasy bazowej i obiekt klasy pochodnej:

W funkcjach składowych i zaprzyjaźnionych klasy pochodnej można **wywoływać nieprywatne funkcje składowe klasy bazowej na rzecz obiektów klasy pochodnej**.

4.3. Konstruktory i destruktory obiektów klasy pochodnej

Obiekt klasy pochodnej zawiera w sobie odziedziczone dane składowe klasy bazowej. Konstruowanie obiektu klasy pochodnej następuje dopiero po uprzednim skonstruowaniu obiektu klasy bazowej. Destrukcja obiektów odbywa się w kolejności odwrotnej do ich konstruowania.

Konstruktory:

Przy tworzeniu obiektów najpierw wywoływany jest konstruktor klasy bazowej, a następnie konstruktor klasy pochodnej.

Destruktory

Przy usuwaniu obiektów najpierw wywoływany jest destruktor klasy pochodnej, a następnie destruktor klasy bazowej.

4.4. Ograniczenia w dziedziczeniu składowych

Nie wszystkie funkcje składowe podlegają dziedziczeniu i mogą być dostępne w klasie pochodnej.

Nie są dziedziczone z klasy bazowej:

- Konstruktory
- Destruktor
- Operator przypisania, funkcja **operator=()**

Konstruktory , destruktor i operator przypisania dla klasy pochodnej powinny zostać w sposób jawny zdefiniowane. W przypadku braku odpowiednich definicji kompilator wygeneruje brakujące funkcje składowe.

ZINTERPRETUJ POSZCZEGÓLNE ELEMENTY KODU ŹRÓDŁOWEGO PONIŻSZEGO PROGRAMU

Publiczna klasa bazowa:

```
#include <iostream>
using namespace std;

//------------------- Klasa bazowa
class pracownik
{
 public:
   char* nazwisko;
   int rok_ur;
   int zarobki;
   static int etaty;           // Liczba zatrudnionych pracowników
   void info();                // Wyświetlenie informacji o obiekcie
   pracownik();                // Konstruktor domniemany
   pracownik(char* n,int r,int z);      // Konstruktor
   ~pracownik();                        // Destruktor
};

//------------------- Konstruktor domniemany klasy pracownik
pracownik::pracownik()
{
   cout<<"Konstruktor domniemany (pracownik)";
   nazwisko="VACAT";
   rok_ur=1900;
   zarobki=100;
   etaty++;
   cout<<" ET="<<etaty<<endl;
}
```

```
//------------------- Konstruktor klasy pracownik
pracownik::pracownik(char* n,int r,int z)
{
   cout<<"Konstruktor (pracownik): "<<n;
   nazwisko=n;
   rok_ur=r;
   zarobki=z;
   etaty++;
   cout<<" ET="<<etaty<<endl;
}

//------------------- Destruktor klasy pracownik
pracownik::~pracownik()
{
   cout<<" Destruktor (pracownik): "<<this->nazwisko;
   etaty--;
   cout<<" ET="<<etaty<<endl;
}

void pracownik::info()
{
   cout<<"INFO pracownik: "<<nazwisko<<" "<<rok_ur
       <<" "<<zarobki<<endl;
}

//------------------- Klasa pochodna z publiczną klasą bazową
class kierownik : public pracownik
{
 public:
   int dodatek;
   kierownik();                          // Konstruktor domniemany
   kierownik(char* n,int r,int z,int d);      // Konstruktor
   ~kierownik();                              // Destruktor
};
```

```
//------------------- Konstruktor domniemany klasy kierownik bez listy inicjalizacyjnej
kierownik::kierownik()
{
   cout<<"Konstruktor domniemany (kierownik)";
   dodatek=150;
   cout<<" ET="<<etaty<<endl;
}

//------------------- Konstruktor klasy kierownik bez listy inicjalizacyjnej
kierownik::kierownik(char* n,int r,int z,int d)
{
   cout<<"Konstruktor (kierownik):"<<endl;
   cout<<"      ----> "<<nazwisko<<" "<<rok_ur
       <<" "<<zarobki<<endl;
   nazwisko=n;
   rok_ur=r;
   zarobki=z;
   dodatek=d;
   cout<<"      ====> "<<nazwisko<<" "<<rok_ur<<" "<<zarobki
       <<" "<<dodatek;
   cout<<" ET="<<etaty<<endl;
}

//------------------- Destruktor klasy kierownik
kierownik::~kierownik()
{
   cout<<" Destruktor (kierownik): "<<this->nazwisko;
   cout<<" ET="<<etaty<<endl;
}

int pracownik::etaty=0;  // definicja i inicjalizacja
                         // skladowej statycznej
```

```
int main()
{
   pracownik a;
   pracownik b("Puchatek",1972,620);
   kierownik c;
   kierownik d("Klapouchy",1965,945,460);
   a.info();
   b.info();
   c.info();          // Błąd, gdyby dziedziczenie niepubliczne
   d.info();          // Błąd, gdyby dziedziczenie niepubliczne
                      // wtedy info() była by niedostępna w funkcji main
}
```

Oczekiwane wyniki:

```
Konstruktor domniemany (pracownik) ET=1
Konstruktor (pracownik): Puchatek ET=2
Konstruktor domniemany (pracownik) ET=3
Konstruktor domniemany (kierownik) ET=3
Konstruktor domniemany (pracownik) ET=4
Konstruktor (kierownik):
      ----> VACAT 1900 100
      ====> Klapouchy 1965 945 460 ET=4
INFO pracownik: VACAT 1900 100
INFO pracownik: Puchatek 1972 620
INFO pracownik: VACAT 1900 100
INFO pracownik: Klapouchy 1965 945
Destruktor (kierownik): Klapouchy ET=4
Destruktor (pracownik): Klapouchy ET=3
Destruktor (kierownik): VACAT ET=3
Destruktor (pracownik): VACAT ET=2
Destruktor (pracownik): Puchatek ET=1
Destruktor (pracownik): VACAT ET=0
```

5. Operatory języka C++

Priorytet	Operator	Wiązanie	Nazwa operatora
1	**( )**	Lewostronne	Wywołanie funkcji
	[ ]	Lewostronne	Indeksowanie
	.	Lewostronne	Wybór składowej
	->	Lewostronne	Wybór wskaźnikowy składowej
	::	Lewostronne	Ustalenie zasięgu
2	**+**	Prawostronne	Jednoargumentowy plus
	-	Prawostronne	Jednoargumentowy minus
	~	Prawostronne	Bitowa negacja
	!	Prawostronne	Logiczna negacja
	++	Prawostronne	Inkrementacja
	--	Prawostronne	Dekrementacja
	*****	Prawostronne	Wyłuskanie
	&	Prawostronne	Pobranie adresu
	sizeof	Prawostronne	Rozmiar
	( )	Prawostronne	Rzutowanie
	new	Prawostronne	Przydzielenie pamięci
	delete	Prawostronne	Zwolnienie pamięci
3	*****	Lewostronne	Mnożenie
	/	Lewostronne	Dzielenie
	%	Lewostronne	Modulo
4	**.***	Lewostronne	Wybór wskazywanej składowej
	->*	Lewostronne	Wskaźnikowy wybór wskazyw. skład.
5	+	Lewostronne	Dodawanie
	-	Lewostronne	Odejmowanie
6	<<	Lewostronne	Bitowe przesunięcie w lewo
	>>	Lewostronne	Bitowe przesunięcie w prawo
7	<	Lewostronne	Relacja mniejszy
	<=	Lewostronne	Relacja mniejszy lub równy
	>	Lewostronne	Relacja większy
	>=	Lewostronne	Relacja większy lub równy
8	==	Lewostronne	Równy
	!=	Lewostronne	Różny
9	&	Lewostronne	Bitowa koniunkcja
10	^	Lewostronne	Bitowa różnica symetryczna
11	|	Lewostronne	Bitowa alternatywa
12	&&	Lewostronne	Logiczna koniunkcja
13	||	Lewostronne	Logiczna alternatywa
14	**?:**	Prawostronne	Warunek

15	=	Prawostronne	Przypisania
	+=	Prawostronne	Złożony przypisania
	-=	Prawostronne	Złożony przypisania
	*=	Prawostronne	Złożony przypisania
	/=	Prawostronne	Złożony przypisania
	%=	Prawostronne	Złożony przypisania
	&=	Prawostronne	Złożony przypisania
	|=	Prawostronne	Złożony przypisania
	<<=	Prawostronne	Złożony przypisania
	>>=	Prawostronne	Złożony przypisania
	^=	Prawostronne	Złożony przypisania
16	,	Lewostronne	Połączenie