Wykład nr 6

Temat: Klasy i abstrakcja danych, cz. II.

Cytaty:

Nade wszystko: bądź szczery wobec siebie.

William Shakespeare, *Hamlet*

Na przyjaciół wybieraj tylko równych sobie.

Konfucjusz

Nasz doktor nigdy nie zabiera się do pracy, jeżeli nie jest to absolutnie konieczne. Taki ma styl.
Na pewno nie zajmie się tobą, jeśli nie potrzebuje pieniędzy.

Herb Shriner

Wykład nr 6

Temat: Klasy i abstrakcja danych, cz. II.

Zakres wykładu:

- wskaźnik „**`this`**”
- konstruktor domniemany
- domyślne kopiowanie składowych
- konstruktor kopiujący (inicjalizator kopiujący)
- konstruktor kopiujący gwarantujący nietykalność
- złożenia: obiekty jako składowe klas
- funkcje i klasy **`friend`** (zaprzyjaźnione)
- funkcja zaprzyjaźniona vs. funkcja składowa klasy
- abstrakcja danych i abstrakcyjne typy danych
- przeciążanie operatorów
- podsumowanie
- ćwiczenia powtórzeniowe i sprawdzające
- następny wykład

Wskaźnik „`this`”

- Każdy obiekt ma dostęp do swego własnego adresu za pomocą wskaźnika (stałego) o nazwie **`this`** (nie jest on jednak częścią samego obiektu).

- Wskaźnik **`this`** jest przekazywany do obiektu (przez kompilator) jako niejawny pierwszy argument każdej niestatycznej funkcji składowej wywoływanej na rzecz tego obiektu.

- Wskaźnik **`this`** jest niejawnie używany do referowania zarówno danych, jak i funkcji składowych obiektu. Istnieje również możliwość jawnego wykorzystania go.

- Typ wskaźnika **`this`** jest zależny od typu obiektu oraz od tego, czy funkcja, w której jest on stosowany, została zadeklarowana jako **`const`** (może być stały wskaźnik do zmiennego obiektu lub stały wskaźnik do stałego obiektu).

- Jednym z ciekawszych zastosowań wskaźnika **`this`** jest uniemożliwienie przypisania obiektu samemu sobie. Przypisanie takie, gdy obiekty zawierają wskaźniki do dynamicznie alokowanych obszarów pamięci, może doprowadzić do wielu poważnych błędów.

- Innym zastosowaniem wskaźnika **`this`** jest umożliwienie kaskadowych wywołań funkcji składowych poprzez zwracanie referencji do obiektu danej klasy (czyli zwracanie wskaźnika `*this`), np. `Klasa &funkcja() {return *this;}`.

```
class Test
{
   int x;                                                  //prywatna dana składowa
public:
   Test(int=0);                                            //konstruktor domyślny
   void drukuj() const;                                    //funkcja składowa klasy
};
////////////////////////////////////////////////////////////////////////
Test::Test(int a)                                          //definicja konstruktora
{
   x=a;
}
////////////////////////////////////////////////////////////////////////
void Test::drukuj() const
{
   cout<<x<<endl;                                          //niejawne użycie wskaźnika this
   cout<<this->x<<endl;                                    //jawne użycie wskaźnika this
   cout<<(*this).x<<endl;                                  //jawne użycie wskaźnika this
}
////////////////////////////////////////////////////////////////////////
main()
{
   Test obiekt(10);                                        //obiekt testowy

   obiekt.drukuj();
}
```

Konstruktor domniemany

- **Konstruktor domniemany** to taki konstruktor, który jest wywoływany bez żadnych argumentów.

- Różnica pomiędzy **konstruktorem z argumentami domyślnymi** a **konstruktorem domniemanym** jest taka, że konstruktor z argumentami domyślnymi **może być** wywołany bez podania żadnych argumentów (**wtedy, gdy ma wszystkie argumenty domyślne**), natomiast konstruktor domniemany **zawsze jest** wywoływany bez podania żadnych argumentów.

- W świetle powyższej definicji konstruktorem, który można wywołać bez żadnych argumentów jest konstruktor ze wszystkimi argumentami domyślnymi, czyli **konstruktorem domniemanym jest jednocześnie konstruktor domyślny!!!**

- Klasa może mieć tylko jeden **konstruktor domniemany** (wynika to z istoty przeciążania funkcji)

```
class Czas
{
   int godzina,minuta,sekunda;                  //prywatne dane składowe
public:
   Czas(void);                                  //konstruktor domniemany
   void ustaw_czas();                           //funkcja składowa klasy
   void drukuj_czas();                          //funkcja składowa klasy
};
////////////////////////////////////////////////////////////////////////
Czas::Czas(void)                                //definicja konstruktora
{
   ustaw_czas();                                //wywołanie funkcji
}
////////////////////////////////////////////////////////////////////////
Czas::ustaw_czas()
{
   //godzina=g;minuta=m;sekunda=s;
}
////////////////////////////////////////////////////////////////////////
main()
{
   Czas czas1;                          //wywołanie bez argumentów!!!
   //lub
   Czas czas1=Czas();                   //wywołanie bez argumentów!!!
}
```

```
class Czas
{
public:
   Czas(char,int,int=0);    //konstruktor z 1 argumentem domyślnym
   Czas(int=0,int=0,int=0);    //konstruktor domyślny (domniemany)
   //Czas(void);               //konstruktor domniemany
};
////////////////////////////////////////////////////////////
```

Domyślne kopiowanie składowych

- Operator przypisania = może być użyty do przypisania jednego obiektu drugiemu tego samego typu.
- Przypisanie to jest domyślnie realizowane jako **kopiowanie składowych** – każda ze składowych obiektu kopiowana jest osobno do tej samej składowej drugiego obiektu.
- Kopiowanie składowych może być przyczyną poważnych problemów, jeżeli klasa wykorzystuje dane składowe, którym dynamicznie przydzielana jest pamięć operacyjna (wskaźniki).

```
class Czas
{
   int godzina,minuta,sekunda;                          //prywatne dane składowe
public:
   Czas(int=0,int=0,int=0);                             //konstruktor domyślny
   void ustaw_czas(int g,int m,int s);                  //funkcja składowa klasy
   void drukuj();                                       //funkcja składowa klasy
};
////////////////////////////////////////////////////////////////////////
Czas::Czas(int g,int m,int s)                           //definicja konstruktora
{
   ustaw_czas(g,m,s);                                   //wywołanie funkcji
}
////////////////////////////////////////////////////////////////////////
Czas::ustaw_czas(int g,int m,int s)
{
   godzina=g;minuta=m;sekunda=s;
}
////////////////////////////////////////////////////////////////////////
Czas::drukuj()
{
   cout<<godzina<<" "<<minuta<<" "<<sekunda<<endl;
}
////////////////////////////////////////////////////////////////////////
main()
{
   Czas czas1;                                  //wszystkie argumenty domyślne
   Czas czas2(13,15,0);                         //podane wszystkie wartości

   czas1=czas2;                                 //przypisanie obiektów
   czas1.drukuj();
}
```

Konstruktor kopiujący (inicjalizator kopiujący)

- **Konstruktorem kopiującym** (ang. *copy constructor*) w danej klasie `Klasa` nazywamy konstruktor, który można wywołać z jednym argumentem typu:

```
Klasa::Klasa(Klasa &);
```

- Argumentem konstruktora kopiującego jest **referencja obiektu** danej klasy.

- **Konstruktor kopiujący MOŻE** mieć też inne argumenty, ale **MUSZĄ** być one domniemane, np:

```
Klasa::Klasa(Klasa &);
Klasa::Klasa(Klasa &,float=3.14,int=1);
```

- **Konstruktor kopiujący** służy do skonstruowania obiektu, który jest kopią innego, już istniejącego obiektu tej klasy, a więc mówimy: „chcę, by nowy obiekt był taki sam jak ten, który posyłam za wzór”.

- **Konstruktor kopiujący** działa na prawach inicjalizacji, a nie przypisania. Oznacza to, że pracuje wtedy, gdy odbywa się **inicjalizacja nowego obiektu**, a nie zwykłe przypisanie.

- **Konstruktor kopiujący** nie jest obowiązkowy. Jeśli go nie zdefiniujemy, wówczas kompilator wygeneruje go sobie sam, ale w najprostszej wersji (zwykłe kopiowanie wartości metodą „składnik po składniku”). **Może to być przyczyną poważnych błędów programu!!!** w przypadku klas zawierających jako dane składowe wskaźniki (obiekty tworzone operatorami **`new`** i **`delete`**)

- **Konstruktor kopiujący** wywoływany jest, gdy:
  a) jawnie tego zażądamy, tj. podczas definicji (inicjalizacji) **nowego obiektu starym**, np.

```
Klasa obiekt_wzor;
//.....
Klasa obiekt_nowy=Klasa(obiekt_wzor);
```

  b) bez naszej wiedzy (niejawne wywołanie konstruktora kopiującego), np.
  - podczas przesyłania argumentów do funkcji, jeśli argumentem funkcji jest obiekt klasy `Klasa`, a przesyłanie odbywa się przez wartość,
  - podczas, gdy funkcja jako swój rezultat zwraca przez wartość obiekt klasy `Klasa`.

Konstruktor kopiujący gwarantujący nietykalność

- „Zwykły” **konstruktor kopiujący** ma prawo modyfikowania obiektu-oryginału (obiektu-wzorca), na wzór którego tworzony jest nowy obiekt. Konstruktor taki ma postać:

```
Klasa::Klasa(Klasa &);
```

- **Konstruktor kopiujący** „gwarantujący nietykalność” to taki konstruktor, który „OBIECUJE”, ŻE NIE ZMIENI OBIEKTU-ORYGINAŁU (obiektu-wzorca), na wzór którego tworzony jest nowy obiekt. Konstruktor odbiera argument jako referencję, ale obiecuje, że obiekt traktuje jako stały. Konstruktor taki ma postać:

```
Klasa::Klasa(const Klasa &);
```

- Sam obiekt wysłany do takiego konstruktora NIE MUSI BYĆ STAŁY!!! (może być obiekt nie stały), jednak cokolwiek wyślemy, to: **KONSTRUKTOR GWARANTUJE NIETYKALNOŚĆ TEGO OBIEKTU!!!**

- ZALECENIE:
  Definiuj **argument konstruktora kopiującego** jako **stałą referencję**!!!

Przykład:

```
class Pieniadz
{
private:
   int banknot;
public:
   Pieniadz(Pieniadz &oryginal);              //konstruktor kopiujący
   //Pieniadz(const Pieniadz &oryginal);      //konstr. kop. gwar. nietyk.
}
////////////////////////////////////////////////////////////////////////
Pieniadz::Pieniadz(Pieniadz &oryginal)
{
   banknot=oryginal.banknot;
   //this->banknot=oryginal.banknot;          //inny zapis
}
////////////////////////////////////////////////////////////////////////

main()
{
   const Pieniadz sto_zlotych;                //stały obiekt
   Pieniadz falsyfikat=sto_zlotych;           //BŁĄD PRZY KOMPILACJI!!!

   //inne zapisy
   //Pieniadz falsyfikat(sto_zlotych);
   //Pieniadz falsyfikat=Pieniadz(sto_zlotych);
}
```

Złożenia: obiekty jako składowe klas

Obiekty jako składowe klas

- **Złożenie** (**agregacja, zawieranie**) (ang. *has a relationship*) polega na tym, że jedna klasa zawiera jako składowe obiekty innych klas. Mówiąc inaczej, obiekt będący całością składa się z określonej liczby obiektów-składników.

Odwieczny problem: być czy mieć?

- Rozróżnienie pomiędzy **dziedziczeniem** a **zawieraniem** może czasami nastręczać pewnych trudności.
- Jeżeli relację pomiędzy dwoma obiektami lepiej opisuje określenie **„ma”** („zawiera”, „składa się” itp.), to należy zastosować **agregację**.
  Kiedy natomiast klasy są naturalnie połączone poprzez stwierdzenie **„jest”**, wtedy odpowiedniejszym rozwiązaniem jest **dziedziczenie**.

Przykład:

`Klasa1` **jest** rodzajem `Klasa2`
`Klasa1` **zawiera obiekt** typu `Klasa2`

`Kwadrat` **jest** rodzajem `Figury`
`Samochód` **zawiera obiekt** typu `Koło`

Konstrukcja obiektu, którego składnikiem jest obiekt innej klasy

- Konstruktory obiektów składowych wykonują się najpierw, a dopiero potem rusza do akcji konstruktor głównego obiektu.
- Obiekt jakiejś klasy będący składnikiem innej klasy **może być inicjalizowany tylko za pomocą listy inicjalizacyjnej!!!**
- Nie można próbować uruchomić konstruktora obiektu składowego z wnętrza konstruktora obiektu głównego.
- Jeśli obiekt składowy nie ma konstruktora, to na liście inicjalizacyjnej nie umieszcza się tego obiektu.
- Jeśli klasa obiektu składowego ma konstruktor wywoływany bez żadnych argumentów (czyli **konstruktor domniemany**), i ten konstruktor chcemy użyć, to można tę pozycję na liście inicjalizacyjnej pominąć.
- Jeśli klasa obiektu składowego ma konstruktory (wszystkie z jakimiś argumentami), to pominięcie wywołania na liście inicjalizacyjnej spowoduje błąd kompilacji.
- Jeżeli jedną ze składowych klasy jest obiekt innej klasy, zadeklarowanie go jako składowej publicznej nie koliduje z hermetyzacją i ukrywaniem informacji dotyczącej prywatnych danych tego obiektu.
- Destruktory wywoływane są w kolejności odwrotnej niż konstruktory. Najpierw rusza do pracy destruktor głównego obiektu (klasy zawierającej obiekty), a dopiero potem wykonywane są destruktory obiektów składowych.

Przykład złożenia obiektów

```
class Volkswagen
{
private:
   Silnik diesel;                                          //obiekt klasy Silnik
   Kolo zapasowe;                                          //obiekt klasy Kolo
   char *nazwa;
   float cena;
public:
   Volkswagen(char *,float,int,int,char *,char *);         //konstr. obiektu głównego
   ...
};
////////////////////////////////////////////////////////////////////////
class Silnik                                    class Kolo
{                                               {
private:                                        private:
   int moc;                                        char *felga;
   int cylindry;                                   char *opona;
public:                                         public:
   Silnik(int=0,int=4);                            Kolo(char* =”brak”,char* =”brak”);
   ...                                             ...
};                                              };
////////////////////////////////////////////////////////////////////////
Volkswagen::Volkswagen(char *model,float ce,int kW,int cyl,char *fel,char *op)
:diesel(kW,cyl),zapasowe(fel,op)                //lista inicjalizacyjna
{
   nazwa=model;
   cena=ce;
}
////////////////////////////////////////////////////////////////////////
main()
{
   Silnik s1;
   Kolo k1;
   Volkswagen Golf(”Golf V”,65499,86,4,”aluminiowa”,”letnia”);   //tworzymy obiekt złożony
   Volkswagen Golf3(”Golf III”,20000,s1,k1);                     //lub w inny sposób ALE należy
}                                                                //dodać nowy konstruktor!!!
```

Funkcje i klasy **`friend`** (zaprzyjaźnione)

- **Funkcja zaprzyjaźniona** z klasą to funkcja, która mimo, że nie jest składnikiem klasy, ma dostęp do wszystkich, **nawet prywatnych!!!** składników klasy.

- Deklarację funkcji zaprzyjaźnionej umieszcza się wewnątrz definicji klasy, do której danych ma mieć dostęp, a nazwę tej funkcji poprzedza się słowem **`friend`**. Słowo to umieszcza się tylko wewnątrz definicji klasy, natomiast nie ma znaczenia, w którym miejscu klasy (przyjacielem albo się jest, albo nie).

- Ważne jest to, że to nie funkcja twierdzi, że jest zaprzyjaźniona, tylko klasa deklaruje, że przyjaźni się z tą funkcją i nadaje jej prawo dostępu do składników prywatnych (**`private`**), chronionych (**`protected`**) i publicznych (**`public`**).

- **Funkcja zaprzyjaźniona** nie jest składnikiem klasy, dlatego też nie posiada wskaźnika **`this`**. Aby odnieść się do składnika klasy, z którą się przyjaźni, musi posłużyć się operatorem '**.**' lub operatorem –> stosując zapis:

```
obiekt.skladnik
wskaznik->skladnik
```

  Nie można pominąć nazwy obiektu, skoro nie ma wskaźnika **`this`**.

- **Funkcja zaprzyjaźniona** może być zwykłą funkcją, a może być też funkcją składową zupełnie innej klasy. Oczywiście ma wtedy dostęp do prywatnych składników swojej klasy, a także tej, z którą się przyjaźni.

```
class Dom_Nowaka
{
private:
   bool kwiatki,telewizor;
public:
   bool podlej_kwiatki() {return kwiatki=true;};
   bool wlacz_telewizor() {return telewizor=true;};
   void stan_domu() {cout<<kwiatki<<" "<<telewizor<<endl;};
   friend void przyjaciel_Nowaka(Dom_Nowaka &);     //deklaracja funkcji friend
};
////////////////////////////////////////////////////////////////////
Dom_Nowaka::Dom_Nowaka()
{  kwiatki=false; telewizor=false;  }
////////////////////////////////////////////////////////////////////
void przyjaciel_Nowaka(Dom_Nowaka &rzecz)  //definicja funkcji friend
{
   //UWAGA: funkcja friend ma dostęp do danych prywatnych!!!
   rzecz.kwiatki=false;                                  //kwiatki nie podlane
   rzecz.telewizor=true;                                 //telewizor włączony
   rzecz.podlej_kwiatki();                               //też dopuszczalne
   //kwiatki=true;                                       //BŁĄD
   //this->kwiatki=true;                                 //BŁĄD
}
////////////////////////////////////////////////////////////////////
main()
{
   Dom_Nowaka dom_pusty;                    //Nowak na wczasach
   dom_pusty.stan_domu();

   przyjaciel_Nowaka(dom_pusty);            //wywołanie funkcji friend
   dom_pusty.stan_domu();
}
```

Funkcja zaprzyjaźniona vs. funkcja składowa klasy

- **Funkcja zaprzyjaźniona** może być przyjacielem więcej niż jednej klasy, czyli może mieć dostęp do prywatnych składników kilku klas.
- Dzięki funkcjom zaprzyjaźnionym możemy nadać dostęp do prywatnych składników klasy nawet takim funkcjom, które są napisane w zupełnie innym języku programowania – np. w Asemblerze, Pascalu czy Fortranie.
- Wykorzystanie funkcji `friend` pozwala na poprawienie prędkości działania programu.
- Funkcje zaprzyjaźnione są bardzo często wykorzystywane do **przeciążania operatorów** używanych z klasami oraz do tworzenia klas iteratorów.
- Jeśli funkcja ma mieć dostęp do składników prywatnych dwóch klas, to masz do wyboru:
  - funkcja jest przyjacielem dwu klas,
  - funkcja jest składnikiem jednej, a przyjacielem drugiej.

Przeciążanie operatorów

Informacje ogólne o przeciążaniu operatorów

- C++ nie pozwala na definiowanie nowych operatorów, a tylko na przeciążanie większości już istniejących. **Przeciążanie operatorów** to przedefiniowywanie tych operatorów, aby poprawnie współpracowały z klasami i aby działały zgodnie z kontekstem, w jakim zostały wykorzystane. Zadaniem kompilatora jest wygenerowanie odpowiedniego kodu na podstawie sposobu ich zastosowania. Jest to jedna z wielu metod rozszerzenia możliwości języka C++.

- Do tej pory operacje na obiektach wykonywane były zwykle za pomocą wysyłania komunikatów do obiektów poprzez wywołania funkcji składowych. Sposób ten może być niewygodny dla pewnych klas obiektów, szczególnie matematycznych. Zastosowanie operatorów ułatwia zrozumienie kodu programu, operacje wykonywane na danych typów zdefiniowanych przez użytkownika (klas) są bardziej zwięzłe.

- Pewne operatory przeciążane są częściej niż inne, szczególnie operator przypisania = oraz niektóre operatory arytmetyczne, jak + czy – (operatory + i –, przeciążone w samym języku C++, działają inaczej na danych typu `int`, `float` czy `double`). Przykładem przeciążonych operatorów bardzo często wykorzystywanych są << (wstawianie do strumienia LUB przesuwanie binarne w lewo zawartości zmiennej) oraz >> (wyciąganie ze strumienia LUB przesuwanie binarne w prawo zawartości zmiennej). Oba operatory zostały przeciążone w bibliotece klas C++.

Szczegóły dotyczące przeciążania operatorów

- Operatory są przeciążane za pomocą definicji funkcji (z nagłówkiem i ciałem) w zwykłej postaci, z wyjątkiem tego, że jako nazwa funkcji wykorzystywane jest słowo kluczowe **`operator`** poprzedzające symbol przeciążanego operatora, np. **`operator+`**.

- Aby móc wykorzystać operator na obiekcie danej klasy, **MUSI** on być przeciążony. Istnieją jednak 2 wyjątki od tej reguły:
  a) operator przypisania = może być użyty z dowolną klasą bez konieczności jawnego przeciążania. Domyślnym trybem jego działania jest **przypisanie składowych** danych składowych klasy (UWAGA na klasy zawierające wskaźniki!!! – dla tych klas należy jawnie przeciążyć operator =).
  b) operator pobierania adresu & również może być użyty bez przeciążania z obiektem dowolnej klasy. Po prostu **zwraca on jego adres w pamięci**. Możliwe jest przeciążenie również tego operatora.

- W celu przeciążenia operatora, programista **MUSI** napisać funkcję przeciążającą tak, aby wykonywała odpowiednie czynności. Czasem lepiej jest, gdy są to **funkcje składowe klasy**, czasem lepiej, gdy **funkcje zaprzyjaźnione** z klasą. Może się też zdarzyć, że są to **zwykłe funkcje globalne**, nie będące ani składowymi, ani zaprzyjaźnionymi z klasą.

Ograniczenia w przeciążaniu operatorów

Operatory, które mogą być przeciążane

```
+       -       *       /       %       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ->*     ,
->      []      ()
new             new[]
delete          delete[]
```

Operatory, które nie mogą być przeciążane

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

- Przeciążanie operatorów nie ma wpływu na kolejność ich wykonywania (w przypadku niejasności kolejności wykonywania przeciążonych operatorów można zawsze zastosować okrągłe nawiasy wymuszające).

- Przeciążanie operatorów nie może zmieniać ich kojarzenia.

- Niemożliwa jest zmiana liczby operandów operatora (przeciążony operator jednoargumentowy w dalszym ciągu pozostaje operatorem jednoargumentowym, a dwuargumentowy dalej dwuargumentowym). Nie może być przeciążany operator trójargumentowy `?:`, natomiast spośród operatorów `&`, `*`, + oraz − każdy ma zarówno wersję jedno-, jak i dwuargumentową, które mogą być przeciążane niezależnie od siebie.

- Nie jest możliwe tworzenie nowych operatorów. Przedefiniowywane mogą być jedynie operatory istniejące.

- Sposób działania operatorów na danych **typów wbudowanych** nie może być modyfikowany za pomocą przeciążania (np. zmiana sposobu działania operatora + na liczbach całkowitych).

- Przeciążanie operatorów działa jedynie z obiektami **typów zdefiniowanych przez użytkownika** lub na różnych połączeniach **typów użytkownika** i **typów wbudowanych**.

- Jeżeli zostanie przeciążony operator przypisania i dodawania, nie oznacza to, że zostanie również przedefiniowany (automatycznie) operator +=, np.

```
obiekt2=obiekt2+obiekt1;   //przeciążony: operator= oraz operator+
obiekt2+=obiekt1;          //należy przeciążyć: operator+=
```

Funkcja operatorowa klasy
vs.
funkcja zaprzyjaźniona

- Funkcje operatorowe, które nie są składowymi klasy, często – ze względów wydajnościowych – deklarowane są jako funkcje zaprzyjaźnione.

- Funkcje składowe, aby pobrać jeden z obiektów klasy będących ich argumentem, posługują się wskaźnikiem **this**. Argument ten w funkcjach, które nie są składowymi (np. w funkcjach zaprzyjaźnionych), **MUSI BYĆ JAWNIE PODANY**.

- Podczas przeciążania operatorów `() [] ->` lub dowolnego z operatorów przypisania, **funkcja operatorowa MUSI BYĆ funkcją składową klasy**. W przypadku pozostałych operatorów nie ma takiego wymogu.

- Dla funkcji operatorowej jako funkcji składowej, skrajny lewy (lub jedyny) operand jest obiektem klasy (lub referencją do obiektu klasy), w której operator jest definiowany.

- Jeżeli skrajny lewy operand musi być obiektem innej klasy lub wbudowanego typu danych, funkcja operatorowa nie może być zadeklarowana jako składowa klasy.

- Funkcja nie będąca funkcją składową musi być zadeklarowana jako funkcja zaprzyjaźniona z klasą, o ile tylko wymaga dostępu do danych składowych prywatnych (**private**) lub chronionych (**protected**).

- Funkcje operatorowe danej klasy wywoływane są jedynie wtedy, gdy lewy operand operatora dwuargumentowego lub jedyny operand operatora jednoargumentowego jest obiektem tej klasy.

- Kolejnym powodem pozwalającym na zdecydowanie, czy funkcja operatorowa nie musi być funkcją składową klasy, jest przemienność operatora, który definiujemy (patrz uwaga powyżej), np.:

```
liczba+duzaCalkowita;   //jako funkcja nie składowa klasy, np. zaprzyjaźniona
duzaCalkowita+liczba;   //jako funkcja składowa klasy
```

gdzie:

```
liczba – np. zmienna (obiekt) typu wbudowanego long int
duzaCalkowita – obiekt typu użytkownika, np. klasy DuzeCalkowite
```

Przykłady

Operator wstawiania do (<<) i pobierania ze (>>) strumienia

```
class Numer_Telefonu
{
private:
   char kod_obszaru[3];       //2-cyfrowy kod obszaru i NULL
   char centrala[4];          //3-cyfrowy numer centrali i NULL
   char linia[5];             //4-cyfrowy numer linii i NULL
public:
   friend ostream & operator<<(ostream &,Numer_Telefonu &);          //funkcja friend
   friend istream & operator>>(istream &,Numer_Telefonu &);          //funkcja friend
};
////////////////////////////////////////////////////////////////////
ostream & operator<<(ostream &wyjscie,Numer_Telefonu &num)
{
   wyjscie<<"("<<num.kod_obszaru<<") "<<num.centrala<<"-"<<num.linia;
   return wyjscie;                                  //umożliwia cout<<a<<b<<c;
}
////////////////////////////////////////////////////////////////////
istream & operator>>(istream &wejscie,Numer_Telefonu &num)
{
   wejscie.ignore();                                //pomiń (   - domyślnie jeden znak
   wejscie>>setw(3)>>num.kod_obszaru;               //wprowadź kod obszaru
   wejscie.ignore(2);                               //pomiń ) i spację
   wejscie>>setw(4)>>num.centrala;                  //wprowadź numer centrali
   wejscie.ignore();                                //pomiń myślnik -
   wejscie>>setw(5)>>num.linia;                     //wprowadź numer linii
   return wejscie;                                  //umożliwia cin>>a>>b>>c;
}
////////////////////////////////////////////////////////////////////
main()
{
   Numer_Telefonu telefon;                          //utworzenie obiektu
   cout<<"Podaj numer telefonu w postaci (12) 345-6789: "<<endl;

   cin>>telefon;              //wywołuje funkcję operator>> jako operator>>(cin,telefon)

   cout<<"Numer tel: "<<telefon;        //wywołuje funkcję jako operator<<(cout,telefon)
}
```

Operatory porównania (== i !=),
przypisania (=), negacji (!), arytmetyczne (*)

```
class Wektor
{
   float x,y,z;
public:
   Wektor(float a=0.0,float b=0.0,float c=0.0):x(a),y(b),z(c) {}
   float zwroc_x() {return x;}
   float zwroc_y() {return y;}
   float zwroc_z() {return z;}
   void ustaw_x(float a) {x=a;}
   void ustaw_y(float a) {y=a;}
   void ustaw_z(float a) {z=a;}
   friend ostream & operator<<(ostream &,Wektor &); //operator 2-argumentowy
   friend bool operator==(Wektor,Wektor);           //operator 2-argumentowy
   friend bool operator!=(Wektor,Wektor);           //operator 2-argumentowy
   Wektor & operator=(Wektor &);                    //operator 2-argumentowy
   Wektor & operator!();                            //operator 1-argumentowy
};
////////////////////////////////////////////////////////////////////////////
Wektor & operator*(Wektor &oryg,float liczba)       //operator 2-argumentowy
{                                                   //jako funkcja globalna
   oryg.ustaw_x(oryg.zwroc_x()*liczba);
   oryg.ustaw_y(oryg.zwroc_y()*liczba);
   oryg.ustaw_z(oryg.zwroc_z()*liczba);
   return oryg;
}
////////////////////////////////////////////////////////////////////////////
ostream & operator<<(ostream &wyj,Wektor &w)        //jako funkcja friend
{
   wyj<<"["<<w.x<<","<<w.y<<","<<w.z<<"]";
   return wyj;
}
```

```
bool operator==(Wektor w1,Wektor w2)                    //jako funkcja friend
{
   if(w1.x==w2.x && w1.y==w2.y && w1.z==w2.z)
      return true;
   else
      return false;
}
////////////////////////////////////////////////////////////////////////////
bool operator!=(Wektor w1,Wektor w2)                    //jako funkcja friend
{  return !(w1==w2);  }
////////////////////////////////////////////////////////////////////////////
Wektor & Wektor::operator=(Wektor &w)                   //jako funkcja składowa
{  x=w.x;y=w.y;z=w.z;
   return *this;  }
////////////////////////////////////////////////////////////////////////////
Wektor & Wektor::operator!()                            //jako funkcja składowa
{  x=-x;y=-y;z=-z;
   return *this;  }
////////////////////////////////////////////////////////////////////////////
main()
{
   Wektor w1,w2(1,2),w3(1,2,3),w4(1,2,3);               //tworzenie wektorów
   cout<<w1<<" "<<w2<<" "<<w3<<" "<<w4<<endl;           //działa operator<<
   w1=w2;                                               //działa operator=
   w2=w2*2;                                             //działa operator* i operator=
   cout<<w1<<" "<<w2<<" "<<w3<<" "<<w4<<endl;
   if(w3==w4) cout<<"Wektory rowne\n"; else cout<<"Wektory rozne\n"; //operator==
   if(w2!=w3) cout<<"Wektory rozne\n"; else cout<<"Wektory rowne\n"; //operator!=
   w2=!w2;                                              //działa operator! i operator=
   cout<<w1<<" "<<w2<<" "<<w3<<" "<<w4<<endl;
}
```

Podsumowanie

Następny wykład

Wykład nr 7

Temat: Projektowanie programów z użyciem techniki orientowanej obiektowo.