Język C++ zajęcia nr 6

Klasy zagnieżdżone

Definicja klasy może być umieszczona wewnątrz definicji innej klasy. Wewnętrzna klasa jest **zagnieżdżona** i ma ograniczony zakres do klasy zewnętrznej. Dostęp do klasy zagnieżdżonej uzyskuje się stosując kwalifikowanie nazwą klasy zewnętrznej.

Zinterpretuj poszczególne elementy kodu źródłowego!!!!

```
#include <iostream>
#include <math.h>
using namespace std;
class odcinek
{
  public:
   //-------------klasa zagnieżdżona ----- początek definicji
   class punkt
   {
   public:
      float x,y;
      void wspolrzedne(float p,float q);
      void wypisz();
   };
   //------------------------------------- koniec definicji
   punkt a,b;
   float dlugosc();
   void wypisz();
};

//-------------------- Kwalifikowanie nazwą klasy zewnętrznej
void odcinek::punkt::wspolrzedne(float p,float q){x=p;y=q;}
void odcinek::punkt::wypisz(){cout<<"("<<x<<";"<<y<<")";}

float odcinek::dlugosc()
{
   return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}

void odcinek::wypisz()
{
   cout<<"[";
   a.wypisz();
   b.wypisz();
   cout<<"]";
}
```

```
int main()
{
   odcinek k;
   k.a.wspolrzedne(2,5);
   k.b.wspolrzedne(6,8);
   k.wypisz();
   cout<<" dlugosc = "<<k.dlugosc();
}
```

Oczekiwane wyniki:

```
[(2;5)(6;8)] dlugosc = 5
```

Uwaga: Niektóre kompilatory przyjmują łagodniejsze zasady dostępu do składowych klasy zagnieżdżonej nadając jej status klasy globalnej.

Zasłaniane definicje `typedef`

Nazwa nadana pewnemu typowi przy pomocy deklaracji **`typedef`** może być zasłonięta przez identyczną nazwę umieszczoną w lokalnej deklaracji **`typedef`** w bloku. Zasięg zasłaniającej deklaracji obejmuje blok, w którym jest ona zawarta.

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

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

int main()
{
   typedef int tygrys;
   {
      typedef char tygrys;
      tygrys x=82;                  // x jest typu char
      cout << x << endl;
   }
   tygrys y=83;                     // y jest typu int
   cout << y;
}
```

```
R
83
```

Wskaźnik `this`

W **funkcji składowej** dostępny jest – w postaci słowa kluczowego **`this`** – **stały wskaźnik** na obiekt, dla którego wywołana została ta funkcja składowa. Zwykle można go pominąć w funkcji składowej przy bezpośrednich odwołaniach do składowych klasy stosując zasadę:

> **Użycie w funkcji składowej nazwy pewnej składowej dotyczy obiektu, dla którego funkcja ta została wywołana.**

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

Pomijanie wskaźnika this:

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

class X
{
  public:
   char* s;
   void drukuj()
   {
      cout << this->s << endl;          // Te dwie instrukcje są
      cout << s << endl;                // sobie równoważne
   }
};

int main()
{
   X a;
   a.s="Krokodyl";
   a.drukuj();
}
```

```
Krokodyl
Krokodyl
```

W niektórych przypadkach użycie wskaźnika **this** jest niezbędne, najczęściej przy bezpośrednim wykorzystaniu wskaźników na obiekty klas.

Zinterpretuj poszczególne elementy kodu źródłowego!!!!

Wykorzystanie wskaźnika this przy tworzeniu listy obiektów:

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

class klient
{
  public:
   char* nazwisko;
   klient* nastepny;          // wskaźnik na następny obiekt w liście
   klient* poprzedni;         // wskaźnik na poprzedni obiekt w liście
   void dolacz(klient& x);    // funkcja dołączająca nowy element
};

void klient::dolacz(klient& x)
//---------- Funkcja dołącza do listy obiekt przekazany przez referencję. Dołączenie
//---------- następuje po obiekcie, na rzecz którego funkcja została wywołana.
{
   x.nastepny=nastepny;
   x.poprzedni=this;
   if (nastepny!=NULL) nastepny->poprzedni=&x;
   nastepny=&x;
}

void drukuj(klient* p)
{
   while(p)
   {
      cout << p->nazwisko << " ";
      p=p->nastepny;
   }
   cout << endl;
}
```

```
int main()
{
   //-------------------- Tworzenie obiektow
   klient a,b,c,d;
   a.nazwisko="Puchatek";
   b.nazwisko="Prosiaczek";
   c.nazwisko="Tygrysek";
   d.nazwisko="Krolik";
   //-------------------- Konstruowanie listy obiektow
   klient* poczatek=&a;
   a.nastepny=NULL;
   drukuj(poczatek);
   a.dolacz(b);
   drukuj(poczatek);
   b.dolacz(c);
   drukuj(poczatek);
   c.dolacz(d);
   drukuj(poczatek);
   //-------------------- Dołączenie dodatkowego obiektu w środku listy po obiekcie b
   klient e;
   e.nazwisko="Klapouchy";
   b.dolacz(e);
   drukuj(poczatek);
}
```

Oczekiwane wyniki:

```
Puchatek

Puchatek Prosiaczek

Puchatek Prosiaczek Tygrysek

Puchatek Prosiaczek Tygrysek Krolik

Puchatek Prosiaczek Klapouchy Tygrysek Krolik
```

Stałe funkcje składowe

Ze stwierdzenia, że **`this`** jest **stałym wskaźnikiem**, wynika zakaz modyfikowania w funkcji składowej samego wskaźnika. Dozwolona jest natomiast modyfikacja wartości danych składowych obiektu wskazywanego przez **`this`**.

Umieszczenie słowa **`const`** po liście parametrów w definicji i deklaracjach funkcji przekształca ją w **stałą funkcję składową** mającą tę własność, że **zabronione jest w niej modyfikowanie obiektu wskazywanego przez `this`** (obiektu, na rzecz którego aktywowana została funkcja składowa, niezależnie czy podczas dostępu do tego obiektu użyjemy, czy też nie użyjemy jawnie wskaźnika **`this`**).

Modyfikator const w definicji stałej funkcji składowej:

```
class liczba
{
  public:
   int plus,minus;
   void przeniesienie(int x)const  // dodane słowo kluczowe const
   {
      plus=plus+x;              // błąd, modyfikacja obiektu zabroniona
      minus=minus-x;            // błąd, modyfikacja obiektu zabroniona
   }
};
```

Konstruktory

Dane składowe klasy umieszczane są w każdym jej obiekcie, więc nie mogą być inicjalizowane przy ich deklaracji wewnątrz definicji klasy. **Dane składowe obiektu mogą być inicjalizowane przez jawnie wywoływane funkcje składowe. Wygodnym sposobem inicjalizowania danych składowych obiektu klasy jest posłużenie się jej konstruktorem.**

Konstruktor jest funkcją składową przeznaczoną do **inicjalizowania obiektów** klas. Nazwa konstruktora jest taka sama jak nazwa klasy. Funkcja ta jest deklarowana i definiowana bez żadnego określenia typu zwracanego wyniku (nawet **`void`**) i nie może zawierać instrukcji **`return`**. Parametry konstruktora nie mogą być typu obiektu klasy, do której należy konstruktor, ale mogą być referencjami do takich obiektów. Konstruktor, podobnie jak inne funkcje składowe, może być zdefiniowany wewnątrz lub na zewnątrz klasy.

> ***Konstruktor jest uruchamiany niejawnie (wywoływany automatycznie przez kompilator) przy definiowaniu obiektu danej klasy.***

Przeciążenie konstruktorów. Klasa może mieć zdefiniowanych wiele konstruktorów różniących się listą parametrów. Wszystkie te konstruktory mają oczywiście identyczne nazwy (takie jak nazwa klasy) i stanowią zespół funkcji przeciążonych.

Konstruktor domniemany. Konstruktor klasy, który może zostać wywołany ***bez argumentów***, jest konstruktorem domniemanym tej klasy. Jeżeli klasa nie ma zdefiniowanego żadnego konstruktora, to kompilator sam wygeneruje konstruktor domniemany dla tej klasy.

Lista inicjalizacyjna konstruktora. W definicji konstruktora, bezpośrednio po jego nagłówku, może wystąpić lista inicjalizacyjna, która jest wykorzystywana do inicjowania danych składowych klasy mających modyfikator **`const`** lub będących obiektami czy referencjami do obiektów innej klasy.

> ***Konstruktor nie przydziela pamięci dla obiektu. Służy jedynie inicjalizowaniu danych składowych klasy.***

Wywołanie konstruktora. Uruchomienie konstruktora następuje automatycznie na skutek użycia w programie konstrukcji składniowej:

nazwa_klasy nazwa_obiektu = nazwa_klasy (lista_argumentów)
lub
nazwa_klasy nazwa_obiektu (lista_argumentów)

Uruchomienie konstruktora jednoargumentowego może nastąpić w wyniku użycia konstrukcji składniowej:

nazwa_klasy nazwa_obiektu = argument

Konstruktory jednoargumentowe określają równocześnie **konwersję konstruktorową** służącą do przekształcania wartości typu argumentu konstruktora do wartości typu klasy konstruktora.

Konstruktory wykorzystywane są w programie również poza definicjami obiektów. Jawne wywołanie konstruktora może wystąpić w wyrażeniu i służyć utworzeniu tymczasowego obiektu.

Destruktory

Destruktor stanowi funkcję składową klasy i jest wywoływany niejawnie (automatycznie przez kompilator) bezpośrednio przed mającą nastąpić likwidacją obiektu tej klasy. Nazwa destruktora to nazwa klasy poprzedzona znakiem **~**. Destruktor nie ma określonego typu (nawet **`void`**), nie zwraca żadnej wartości i nie ma parametrów. Nie może być przeciążony.

Destruktor nie likwiduje obiektu i nie zwalnia zajmowanej przez niego pamięci. Może wykonywać pomocnicze czynności przed zlikwidowaniem obiektu.

Destruktor jest uruchamiany niejawnie (wywoływany automatycznie przez kompilator) przy likwidowaniu obiektu danej klasy.

Likwidowanie obiektu następuje na skutek:

- Zakończenia programu (zakończenie funkcji **`main`**).
- Wyjścia z bloku, w którym powstał obiekt lokalny nie statyczny.
- Wykonania operatora **`delete`** (tylko po wcześniejszym **`new`**).

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

```
#include <iostream>
#include <string.h>
using namespace std;

class pociag
{
  public:
    char odjazd[15];
    char dokad[20];
    pociag(char *gd, char *dd);     // Konstruktor dwuargumentowy
    pociag(char *gd);               // Konstruktor jednoargumentowy
    pociag();                       // Konstruktor domniemany
    ~pociag();                      // Destruktor
};
pociag::pociag(char *gd, char *dd)
{
  strcpy(odjazd,gd);
  strcpy(dokad,dd);
  cout << "\n+++Konstruktor dwuargumentowy: Powstaje pociag "
       << dokad << ", odjazd " <<odjazd;
}
pociag::pociag(char *gd)
{
  strcpy(odjazd,gd);
  strcpy(dokad,"w nieznane");
  cout <<"\n+++Konstruktor jednoargumentowy: Powstaje pewien "
         "pociag, odjazd " <<odjazd;
}
pociag::pociag()
{
  strcpy(odjazd,"nie ustalony");
  strcpy(dokad,"rezerwowy");
  cout << "\n+++Konstruktor domniemany: Powstaje pociag bez "
         "ustalonych parametrow";
}
pociag::~pociag()
{
  cout << "\n---Destruktor: Likwidowany jest pociag " << dokad
       << ", odjazd " <<odjazd;
}
void jaki_pociag(pociag &poc)
{
  cout << "\nTo jest pociag " << poc.dokad << " odjazd "
       << poc.odjazd;
}
```

```
int main()
{
  pociag a = pociag("16:20", "do Kalisza");
  pociag b("9:48");
  pociag c;
  jaki_pociag(a);
  jaki_pociag(b);
  jaki_pociag(c);
  cout << "\n========== Wejscie do bloku ==========";
  {
    pociag a("12:35", "do Szczecina");
    pociag b;
    jaki_pociag(a);
    jaki_pociag(b);
  }
  cout << "\n========== Wyjscie z bloku ===========";
  jaki_pociag(a);
  jaki_pociag(b);
}
```

Oczekiwane wyniki:

```
+++Konstruktor dwuargumentowy: Powstaje pociag do Kalisza, odjazd
16:20
+++Konstruktor jednoargumentowy: Powstaje pewien pociag, odjazd 9:48
+++Konstruktor domniemany: Powstaje pociag bez ustalonych parametrow
To jest pociag do Kalisza odjazd 16:20
To jest pociag w nieznane odjazd 9:48
To jest pociag rezerwowy odjazd nie ustalony
========== Wejscie do bloku ==========
+++Konstruktor dwuargumentowy: Powstaje pociag do Szczecina, odjazd
12:35
+++Konstruktor domniemany: Powstaje pociag bez ustalonych parametrow
To jest pociag do Szczecina odjazd 12:35
To jest pociag rezerwowy odjazd nie ustalony
---Destruktor: Likwidowany jest pociag rezerwowy, odjazd nie
ustalony
---Destruktor: Likwidowany jest pociag do Szczecina, odjazd 12:35
========== Wyjscie z bloku ===========
To jest pociag do Kalisza odjazd 16:20
To jest pociag w nieznane odjazd 9:48
---Destruktor: Likwidowany jest pociag rezerwowy, odjazd nie
ustalony
---Destruktor: Likwidowany jest pociag w nieznane, odjazd 9:48
---Destruktor: Likwidowany jest pociag do Kalisza, odjazd 16:20
```