C++ - klasy

Typy definiowane przez użytkownika

W C++ dane mogą zostać powiązane z funkcjami – za pomocą własnego typu danych

Ten typ to nie tylko jedna lub kilka zebranych razem liczb, to także sposób ich zachowania jako całości

Tak zdefiniowany typ może być modelem rzeczywistego obiektu

Mówimy ***typ*** dlatego, że kreujemy nie jeden konkretny obiekt, ale ***klasę*** obiektów

Klasa to typ – tak jak typem jest int, float.

Definicja klasy

Definicja klasy:

```
class Moja_klasa
{
        // ciało klasy
};      //koniec definicji klasy (ze średnikiem !)
```

Utworzenia konkretnego obiektu **Moja_klasa** (zmiennej typu **Moja_klasa**):

```
Moja_klasa obiekt1;     //utworzenie 1 egzemplarza obiektu (instancji)
                        //klasy Moja_klasa
Moja_klasa obiekt_2;    //utworzenie 2 egzemplarza obiektu
                        //(instancji) klasy Moja_klasa
```

Przykład klasy

```
class samochód {
                        // ciało klasy
                    };
samochód s1, s2;        //utworzenie 2 obiektów klasy
                                //samochód
```

```
samochód *wsk_sam;          //utworzenie wskaźnika do
                            //obiektu klasy samochód
samochód &ref_sam=s1; //utworzenie referencji do
                              //obiektu klasy samochód
```

Składniki klasy

W ciele klasy deklaruje się **składniki klasy**, którymi mogą być:

pola klasy (atrybuty klasy - dane składowe) - różnego typu dane (int, float, itp.)

metody klasy (funkcje składowe)

Aby odnieść się do pola obiektu można posługiwać się jedną z notacji:

obiekt.składnik

wskaźnik->składnik

referencja.składnik

Operator '.' (kropka) – odniesienie do składnika obiektu znanego z nazwy lub referencji

Operator '->' - odniesienie do składnika obiektu pokazywanego wskaźnikiem

Przykład – odwołanie do pola klasy

```
class samochód {
        public: int rok_pr;
                    float cena;
                    char marka[20];
                    ...
                };
samochód s1;          //definicja egzemplarza obiektu
samochód *wsk_sam;          //definicja wskaźnika
samochód &ref_sam=s1;       //definicja referencji
                                    //odwołanie do składnika cena:
```

s1.cena=35000; //z obiektu s1

wsk_sam=&s1;

wsk_sam->cena=35000; //ze wskaźnika do obiektu

ref_sam.cena=35000;//z referencji do obiektu

Składniki klasy (pola i metody)

Składnikami klasy mogą też być funkcje – funkcje składowe klasy.

Deklaracje funkcji mogą być pomieszane z deklaracjami danych

Niezależnie od miejsca zdefiniowania składnika wewnątrz klasy – składnik znany jest w całej definicji klasy, tzn. nazwy deklarowane w klasie mają zakres ważności równy obszarowi całej klasy

Klasa w przeciwieństwie do funkcji nie ma początku ani końca – to pudło na składniki

Składnikiem klasy może być dana typu wbudowanego (int, float, char) ale też obiekt typu zdefiniowanego przez użytkownika np. obiekt innej klasy

Np. obiekt klasy samochód ma silnik, który może być obiektem innej klasy definiującej parametry i funkcje silnika.

Przykład – definicja klasy z polami i metodami

```
class samochód {
        public: int rok_pr;
                        float cena;
                        char marka[20];
                        //pozostałe dane
                        //…
                        //metody klasy:
                        void hamuj();
                        void jedz_prosto();
                        //znowu jakieś dane
                        //…
```

```
    //znowu metody klasy
    //…
};
```

Paradygmaty programowania obiektowego

Abstrakcja - każdy obiekt jest modelem abstrakcyjnym, który może wykonywać pracę, opisywać i zmieniać swój stan oraz komunikować się z innymi obiektami bez ujawniania, w jaki sposób zaimplementowano dane cechy.

Hermetyzacja - czyli ukrywanie implementacji (enkapsulacja) - zapewnia, że obiekt nie może zmieniać stanu wewnętrznego innych obiektów w nieoczekiwany sposób - tylko własne metody obiektu są uprawnione do zmiany jego stanu

Polimorfizm – referencje i kolekcje obiektów mogą dotyczyć obiektów różnego typu, a wywołanie metody dla referencji spowoduje zachowanie odpowiednie dla pełnego typu obiektu wywoływanego. Jeśli dzieje się to w czasie działania programu, to nazywa się to *późnym wiązaniem* lub *wiązaniem dynamicznym*. Niektóre języki udostępniają bardziej statyczne (w trakcie kompilacji) rozwiązania polimorfizmu – na przykład szablony i przeciążanie operatorów (w C++)

Dziedziczenie – umożliwia definiowania i tworzenia klas pochodnych - nie trzeba redefiniować całej funkcjonalności klasy, lecz tylko tę, której nie ma klasa bazowa

Enkapsulacja

W ciele klasy dane i metody zamknięte są jak w kapsule (ang. encapsulation)

Definicja klasy sprawia, że dane i funkcje, które były luźno rozrzucone w programie strukturalnym – w programowaniu obiektowym zamknięte są w kapsule klasy

Różne klasy mogą posiadać tak samo nazwane funkcje a i tak kompilator będzie wiedział jaką funkcję wywołać dla odpowiednich obiektów

Kapsuła może być przezroczysta lub nie - coś może być dostępne spoza klasy lub nie (zależy to od specyfikatorów dostępu)

Specyfikatory dostępu

Są trzy rodzaje dostępu do składnika klasy:

składnik private – dostępny tylko dla funkcji składowych danej klasy (dla danych składowych oznacza to, że tylko funkcje będące składnikami klasy mogą te prywatne dane odczytywać lub do nich

zapisywać. Dla funkcji oznacza to, że mogą one zostać wywołane tylko przez inne funkcje składowe tej klasy

składnik protected – tak jak private, ale także dla klas wywodzących się od tej klasy (dziedziczących po tej klasie)

składnik public – dostępny bez ograniczeń (składniki klasy mogą być używane z jej wnętrza a także spoza zakresu klasy)

Przez domniemanie składniki mają dostęp **private**.

Etykiety powyższe można umieszczać w dowolnej kolejności, mogą się też powtarzać, lepiej jednak nie mieszać i składniki o danym dostępie pogrupować razem.

Klasa a obiekt Funkcje składowe klasy

Klasa – to nowy typ zmiennej

Mając zdefiniowaną klasę – można tworzyć obiekty danej klasy

Sama definicja klasy nie definiuje żadnych obiektów. Klasa to typ obiektu a nie sam obiekt.

Funkcje zadeklarowane wewnątrz klasy są funkcjami składowymi (metodami) tej klasy. Mają zakres klasy w której je zadeklarowano. Mają pełny dostęp do wszystkich składników klasy (danych - własności i innych funkcji klasy). Dostęp do funkcji klasy realizuje się dla poszczególnych obiektów klasy (podobnie jak do pól składowych klasy).

Definicja metod (funkcji) klasy

Sposoby (miejsca) definiowania funkcji klasy:

Wewnątrz definicji klasy

Poza ciałem klasy

Oba sposoby definiują funkcję o zakresie ważności klasy. Różnica dla kompilatora:

Sposób 1 - kompilator uznaje, że funkcja jest typu inline

Sposób 2 - kompilator uznaje, że funkcja nie jest typu inline, ale może być jeśli to wyraźnie zaznaczymy np.
inline void student::wyswietl()

Przykład – definicja metod w ciele klasy

class student{ char nazwisko[80];

```
                    int wiek;
public: // *** definicje funkcji w ciele klasy ***
void wczytaj(char *napis, int lata)
{       strcpy(nazwisko,napis);
        wiek=lata;
}
void wyswietl()
{       cout<<nazwisko<<" lat:"<<wiek<<endl;
}
 };     //koniec definicji klasy
void main()
{       student s1,s2,s3;
        s1.wczytaj("Kowalski",20); s1.wyswietl();
        s2.wczytaj("Malinowski",21); s2.wyswietl();
        s3.wczytaj(„Wisniewski”,24); s3.wyswietl();
}
```

Przykład 1 – definicja metod poza ciałem klasy

```
class student {char nazwisko[80];
                    int wiek;
                    public: void wczytaj(char *napis, int lata); //prototyp funkcji
                    void wyswietl();//prototyp funkcji
};      //koniec definicji klasy
// *** definicja funkcji poza cialem klasy ***
void student::wczytaj(char *napis, int lata)
{       strcpy(nazwisko,napis);
```

```
	wiek=lata;
}
void student::wyswietl()
{	cout<<nazwisko<<" lat:"<<wiek<<endl;
}
void main()
{	student s1; s1.wczytaj("Kowalski",20); s1.wyswietl(); }
```

Definicja funkcji poza ciałem klasy

Definiując funkcję sposobem 2 - w ciele klasy umieszcza się **prototyp funkcji** a definicję funkcji **poza ciałem klasy**

Nazwa funkcji poza ciałem klasy jest uzupełniana nazwą klasy i operatorem zakresu **::**

Wskaźnik this

Funkcja składowa wykonuje operacje na danych składowych klasy

Funkcję wywołujemy na rzecz konkretnego obiektu klasy tzn.:
obiekt.funkcja_klasy(parametry);

Bez naszej wiedzy do wnętrza funkcji przesyłany jest wskaźnik (z adresem), do tego konkretnego obiektu, na rzecz którego wołamy funkcję klasy. Tym adresem funkcja inicjalizuje sobie swój specjalny wskaźnik zwany **this (ang. ten)**

Wskaźnik ten pokazuje funkcji, na którym konkretnie obiekcie tej klasy ma funkcja pracować

Przykład – wskaźnik this

Na przykład funkcję wczytaj() z klasy Student:

```
void wczytaj(char *napis, int lata)
{	strcpy( nazwisko, napis );
	wiek = lata;
}
```

Można napisać w postaci:

```
void student::wczytaj( char *napis, int lata)
{       strcpy( this->nazwisko,napis );
        this->wiek = lata;
}
```

Wskaźnik this

Brak wskaźnika this - kompilator sam sobie uzupełnia, oszczędzając nam pisania.

Zwykły wskaźnik mogący pokazywać na obiekty klasy XXX ma typ:

XXX *wsk;

Wskaźnik this pokazuje na takie obiekty, ale nie można nim poruszać, czyli jest typu:
XXX const *wsk;

Zasłanianie nazw

Nazwy składników klasy mają zakres klasy a więc w obrębie klasy zasłaniają elementy o takiej samej nazwie leżące poza klasą

```
int kolor=1;
class punkt {
  int x,y;
  public: int kolor;
void ustaw_wspolrzedne(int a, int b)
{
this->x=a; //lub x=a;
this->y=b; //lub y=b;
}
```

```
 };
main()
{       punkt a,b;
        a.kolor=2;
        b.kolor=4;
        cout<<"Zmienna kolor nie mająca nic wspólnego z klasą:"<<kolor<<endl;
        cout<<"Punkt a w kolorze:"<<a.kolor<<", punkt b w kolorze: " <<b.kolor<<endl;
        a.kolor=3;
        b.kolor=6;
        kolor=8;
}
```

Konstruktor

Definicję obiektu i nadanie mu wartości można wykonać za pomocą specjalnej funkcji klasy zwanej konstruktorem

Konstruktor:

nazywa się tak samo jak klasa

przed konstruktorem nie ma określenia żadnego typu wartości zwracanej (nawet typu void) - dlatego w konstruktorze nie może wystąpić instrukcja return

Przykład konstruktora

```
class punkt {
        int x,y;
public:
        punkt(int a, int b) //konstruktor klasy
        {       x=a;
                y=b;
        }
```

```
    void wyswietl()
    {
        cout<< "(" << x << "," << y << ")";
    }
};
```

Konstruktor

Wywołanie:

```
    punkt p1=punkt (2,4); //utworzenie obiektu klasy punkt i                    //nadanie
mu poprzez konstruktor
                          //współrzędnych x=2, y=4
    punkt p2(2,4);        //mniej czytelnie - opuszczono nazwę
                          //konstruktora (identyczną z nazwą klasy)
```

Osobliwością konstruktora jest to, że jest on wywoływany automatycznie ilekroć tworzymy obiekt danej klasy

Konstruktor nie konstruuje obiektu ale nadaje mu wartość początkową

Jest uruchamiany tylko raz - gdy obiekt jest powoływany do życia.

Nie jest obowiązkowy ale wygodny

Konstruktor to funkcja dla której najczęściej spotyka się przeładowanie nazwy

Przeładowane konstruktory

Klasa:

```
class punkt{
    int x,y;
public:
punkt() {x=0;y=0;}          //konstruktor bezargumentowy
```

punkt(int a) {x=a;y=a;} //konstruktor z jednym argumentem

punkt(int a, int b) {x=a;y=b;} //konstruktor z dwoma arg.

void wyswietl() { cout<<"("<<x<<","<<y<<")";}

};

Tworzenie obiektów klasy punkt:

punkt p1, p2(2), p3(4,5);

p1.wyswietl();

p2.wyswietl();

p3.wyswietl();

Definicja konstruktora poza ciałem klasy

```
class punkt{
        int x,y;
public:
punkt();                //konstruktor bezargumentowy
punkt(int a);   //konstruktor z jednym argumentem
void wyswietl() { cout<<"("<<x<<","<<y<<")";}
};


        punkt::punkt() {x=0;y=0;}
        punkt::punkt(int a){x=a;y=a;}
```

Destruktor

Przeciwieństwo konstruktora

Funkcja składowa klasy wywoływana wtedy gdy obiekt danej klasy ma być zlikwidowany

Definicja destruktora:

```
~nazwa_klasy()
{ ... }
```

Destruktor jest bezargumentowy i również nie zwraca żadnej wartości

Nie jest konieczny

Konstruktor i destruktor

Przykład 1

```
class punkt
{
        int x,y;
 public:
        punkt(int,int); //konstruktor
        ~punkt();      //destruktor
        void wyswietl();

};      //koniec definicji klasy
```

Przykład 1 (c.d.)

```
// *** definicja konstruktora poza cialem klasy ***
punkt::punkt(int k, int l)
{       x=k; y=l; cout<<"\nTworze obiekt punkt";
}
// *** definicja destruktora poza cialem klasy ***
punkt::~punkt()
{       cout<<"\nUsuwam obiekt klasy punkt";
```

```
}
// *** definicja metody poza cialem klasy ***
 void punkt::wyswietl()
{ cout<<"("<<x<<","<<y<<")";}
```

Przykład 1 – test klasy w main

```
punkt p(1,2);  //obiekt globalny
void main()
{ cout<<"\nFunkcja main";
  punkt p1(1,2); cout<<"\nP1";p1.wyswietl();
  punkt *p2=new punkt(3,4);
  cout<<"\nP2"; p2->wyswietl();
  delete(p2);
  p2=new punkt(15,15);  cout<<"\nP2";p2->wyswietl();
  delete(p2);
  punkt p3(2,3);  cout<<"\nP3";p3.wyswietl();
  punkt &p=p3;
  cout<<"\n&P=p3";p.wyswietl();
  cout<<"\nKoniec main";
}
```

Przykład 1
– wynik

Statyczny składnik klasy

Dana określona jako statyczna jest w pamięci tworzona jednokrotnie i jest wspólna dla wszystkich obiektów danej klasy

Istnieje już gdy nie zdefiniowano jeszcze żadnego obiektu tej klasy

Dzięki składnikom statycznym można zmniejszyć liczbę potrzebnych zmiennych globalnych

Składnik statyczny dotyczy nie poszczególnych obiektów danej klasy, ale samej klasy jako całości

Definicja pola statycznego klasy

Poprzedzona słowem kluczowym **static,** np.:

```
class punkt
{       int x,y;
        public:
          static int ile; //składowa statyczna klasy
          //pozostałe składniki klasy
};
```

Definicja składowej statycznej musi być umieszczona, tak jak definicja zmiennej globalnej (może być również zainicjalizowana wartością początkową) poza ciałem klasy:

```
int punkt::ile=0; //inicjalizacja składnika statycznego
                  //poza ciałem klasy
```

Pole statyczne może być także private i można je inicjalizować w powyższy sposób, ale później nie ma już do niego dostępu spoza klasy (tak jak dla zwykłych danych private)

Odniesienie do pola statycznego

Za pomocą nazwy klasy i operatora zakresu:
klasa::pole_statyczne;

Za pomocą obiektu klasy (jeśli już istnieje)
obiekt.pole_statyczne;

Za pomocą wskaźnika do obiektu klasy:

```
klasa obiekt, *wsk;
wsk=&obiekt;
wsk->pole_statyczne;
```

tak jak dla zwykłych pól klasy (nie statycznych)

Pole statyczne - uwagi

Prywatne pole statyczne można zainicjalizować poza klasą

Domyślnie pole statyczne (podobnie jak zmienne globalne) inicjalizowane jest automatycznie wartością 0

Klasy, które są definiowane lokalnie (wewnątrz innej klasy) nie mogą mieć składników statycznych

Pole statyczne może pojawić się jako argument domniemany w metodzie tej klasy (zwykłe pole nie może)

Statyczne metody klasy

Metoda statyczna pracuje tylko na składowych statycznych klasy (polach i ewentualnie na innych metodach statycznych)

Metodę statyczną można wywołać nie tylko z obiektu danej klasy ale także z samej klasy (poprzez jej nazwę)

Wewnątrz metody statycznej nie jest możliwe jawne odwoływanie się do wskaźnika **this** (wskaźnik this odnosi się do konkretnego obiektu klasy a nie do samej klasy) ani do pól nie statycznych

Dotyczy nie konkretnego obiektu, ale całej klasy i można ją wywołać jeśli jeszcze nie istnieje żaden obiekt danej klasy:
klasa::metoda_statyczna();

Po utworzeniu obiektu klasy i wskażnika do obiektu można ją wywołać również:
obiekt.metoda_statyczna();
wsk->metoda_statyczna();

Metody składowe typu const

Metoda **const** to taka funkcja, która nie modyfikuje danych składowych obiektu, na rzecz którego jest wywołana

Jeśli obiekt danej klasy jest **const** to na jego rzecz można wywoływać jedynie te metody składowe, które są **const**

Np.:

```
const float pi=3.1416;
const punkt p0(0,0);
```

Przykład 2 – składniki statyczne

```
class punkt{
        int x,y;
 public:
        static int ile;  //pole statyczne klasy
        punkt(int,int); //konstruktor
        ~punkt();       //destruktor
        void wyswietl();
        static void wyswietl_ile()   //metoda statyczna
        //musi byc definicja w ciele klasy
        {       cout<<"\nIlosc obiektow:"<<ile;
        }
};      //koniec definicji klasy
```

Przykład 2 – c.d.

```
punkt::punkt(int k, int l)
{       x=k;    y=l;    ile++;
        cout<< "\n Tworze obiekt punkt";
}
punkt::~punkt()
{       cout<< "\n Usuwam obiekt klasy punkt";
        ile--;
```

```
}
 void punkt::wyswietl()
{ cout<<"("<<x<<","<<y<<")";
}
```

Przykład 2 (c.d.)

```
int punkt::ile=0; //deklaracja pola statycznego poza klasą
void main()
{ cout<<"\nFunkcja main";
  punkt::wyswietl_ile();
  punkt p1(1,2);   cout<<"\nP1";p1.wyswietl();
  p1.wyswietl_ile();
  punkt *p2=new punkt(3,4);  cout<<"\nP2";p2->wyswietl();
  p2->wyswietl_ile();
  delete(p2);
  punkt::wyswietl_ile();
  p2=new punkt(15,15);  cout<<"\nP2";p2->wyswietl();
  delete(p2);
  punkt p3(2,3);  cout<<"\nP3";p3.wyswietl();
  punkt &p=p3;  cout<<"\n&P=p3";p.wyswietl();
  p.wyswietl_ile();
  cout<<"\nKoniec main";      }
```

Destruktor - uwagi

Destruktor można wywołać jawnie spoza klasy:

obiekt.~klasa();

wskaznik->~klasa();

Destruktor uruchamiany z wnętrza klasy:

this->~klasa();

Jawne wywołanie destruktora nie likwiduje obiektu - obiekt ciągle istnieje, jedynie został zmodyfikowany

Zastosowanie destruktorów np. praca z plikami (konstruktor – otwarcie pliku, destruktor – zamknięcie pliku), dynamiczny przydział pamięci (konstruktor – rezerwacja pamięci operatorem new, destruktor – zwolnienie przydzielonej pamięci operatorem delete)

Konstruktor - uwagi

Konstruktor domniemany - można wywołać beż żadnych argumentów (nie koniecznie jest to konstruktor bezargumentowy), czyli ze wszystkimi argumentami domniemanymi

Klasa może mieć tylko jeden konstruktor domniemany, np.:

```
class wektor3w{
        float x,y,z;
    public:
        wektor3w(float a=0.0, float b=0.0, float c=0.0)
    { x=a;  y=b;    z=c;}
};
```

Gdy klasa nie ma ani jednego konstruktora, to kompilator wygeneruje sobie dla tej klasy konstruktor domniemany (public)

Definicja stałego pola klasy

Definicja zwykłej stałej:

const int N = 100; //określa obiekt stały o nazwie **N** i wartości 100

Obiekty stałe mogą być tylko inicjalizowane, nie wolno do nich nic przypisywać - inicjalizacja to nadanie wartości w momencie narodzin

W klasie:

```
class abc{
```

```
        const int N;
        // ...
};
```

W ciele klasy nie wolno inicjalizować składnika stałego w taki sposób:

```
class abc{
        const int N=100;        //źle!!!
        // ...
};
```

Do zainicjalizowania składnika stałego służy konstruktor, a konkretnie **lista inicjalizacyjna konstruktora**

Konstruktor z listą inicjalizacyjną

```
klasa::klasa(argumenty):lista_inicjalizacyjna
{
        //ciało konstruktora
}
```

Np.:

```
class abc{
 const int N;
 float x; char c;
 abc(float liczba, int rozmiar, char znak):N(rozmiar),c(znak)
        {       x=liczba;}
};
```

lub:

```
class abc{
        const int N; float x; char c;
        abc(float liczba, int rozmiar, char znak);
```

```
};
abc::abc(float liczba,int rozmiar, char znak):N(rozmiar),c(znak)
        {       x=liczba;       }
```

Wykonanie konstruktora składa się z 2 etapów:

inicjalizacja składników (wykonywany dzięki liście inicjalizacyjnej)

przypisania i inne akcje

składnikowi nie const możemy w konstruktorze nadać wartość na dwa sposoby:

przez listę inicjalizacyjną konstruktora

przez zwykłe podstawienie w ciele konstruktora

składnikowi **const** można nadać wartość początkową tylko za pomocą listy inicjalizacyjnej.

lista inicjalizacyjna nie może inicjalizować składnika **static**