C++: Klasy i obiekty I

Podstawowe (wbudowane) typy danych

Jak wiemy, język C daje do dyspozycji programisty pewną ilość tak zwanych wbudowanych typów danych jak na przykład `int`, `float`, `char`.

Zadeklarowanie zmiennej jakiegoś typu powoduje zarezerwowanie w pamięci odpowiedniej ilości bajtów w celu przechowywania wartości tej zmiennej. Na przykład deklaracja

```
int i;
```

powoduje zarezerwowanie czterech bajtów i skojarzenie nazwy zmiennej `i` z ich adresem

Natomiast instrukcja

```
i=2002;
```

powoduje umieszczenie pod tym adresem liczby 2002.

Gromadzenie danych

Załóżmy, że mamy napisać program dla salonu sprzedaży samochodów, który będzie ewidencjonował cechy samochodów znajdujących się aktualnie w ofercie. Chodzi o cechy takie jak marka, kolor, pojemność silnika itd.

Jeśli mamy do dyspozycji tylko wbudowane typy danych, to zazwyczaj deklarujemy odpowiednią ilość tablic, na przykład

```
.....
int kolor[30];
int pojemnoscSilnika[30];
.....
```

i odpowiednią cechę czwartego samochodu wyczytujemy z czwartego elementu stosownej tablicy.

Takie podejście ma fundamentalną wadę. W rzeczywistym świecie, gdy patrzymy na konkretny egzemplarz samochodu, na przykład wystawiony przed salonem, od razu rozpoznajemy jego

markę i kolor, a pojemność silnika możemy sprawdzić zaglądając pod jego maskę, a nie do ewidencji na zapleczu salonu.

Natomiast w naszym programie informacje o konkretnych samochodach rozrzucone są po oddzielnych, niczym nie powiązanych ze sobą tablicach.

O wiele wygodniej byłoby dysponować specjalnym typem danych samochod, w którym można byłoby zgromadzić wszystkie potrzebne nam cechy samochodu.

Język C++ umożliwia programiście tworzenie własnych typów danych, odpowiednio dostosowanych do potrzeb tworzonego programu.

Abstrakcja

Człowiek nie myśli o samochodzie jako o tworze złożonym z kilku milionów części. Jest on dla niego samodzielną dobrze określoną jednostką, która nadaje się do spełnienia pewnych funkcji jak wyjazd na miesięczne wakacje czy przywiezienie zakupów ze sklepu.

Nawet mechanik samochodowy nie myśli o wszystkich najdrobniejszych częściach, jakie można znaleźć w samochodzie. Myśli on w kategoriach kilkunastu, najwyżej kilkudziesięciu podzespołów, które muszą dobrze działać, aby samochód mógł sprawnie i bezpiecznie jeździć. O tym, że podzespoły składają się z mniejszych elementów myśli się dopiero wtedy, gdy podzespół nie działa i trzeba go naprawić.

Analogicznie postępujemy w programowaniu obiektowym. O problemie myślimy w kategoriach obiektów mających pewne cechy, czy zbudowanych z pewnej ilości składników. Nie rozważamy wszystkich rzeczywistych cech i rzeczywistych składników, a jedynie te, które są istotne w danym zagadnieniu. W ten sposób dochodzimy do potrzebnego nam nowego typu danych.

Abstrakcja to pierwsza cecha charakterystyczna programowania obiektowego. Obejmuje ona w szczególności (ale nie tylko) możliwość tworzenia nowych typów danych.

Pojęcie klasy i obiektu

- Definiowanie nowych typów realizuje się poprzez tak zwane **klasy**. Klasa to abstrakcyjna definicja jeszcze nie istniejącego obiektu, określająca jakie cechy charakteryzują dany obiekt i jakim operacjom obiekt ten można poddawać.
- Gdy mamy zdefiniowaną klasę, powołujemy do życia obiekty podając konkretne wartości cech jaki dany egzemplarz wyróżniają.
- Na bazie jednej klasy na ogół powołujemy do życia wiele obiektów.
- Dwa obiekty tej samej klasy mają zawsze ten sam zestaw cech, ale konkretne wartości przypisane tym cechom mogą być różne.

Z punktu widzenia formalnego klasa to recepta na tworzenie nowego, własnego typu danych połączona z instrukcją w jaki sposób można dane tego nowego typu przetwarzać.

Definicja klasy obejmuje

- dane składowe z których zbudowany jest nowy typ.
- funkcje składowe, które określają jakie operacje na danych składowych można wykonywać

- W programowaniu strukturalnym nagina się istniejący świat do komputera.
- W programowaniu obiektowym nagina się komputer do istniejącego świata.

Definiowanie klasy

W C++ do definiowania klas używa się słów kluczowych `struct` lub `class`.

Na początek łatwiejsze jest definiowanie klas przy użyciu **struct**.

Na przykład

```
struct samochod{
    char model[20];
    float przebieg;
```

```
    void jedz(float ile_kilometrow){
      przebieg=przebieg+ile_kilometrow;
    };
};
```

definiuje klasę `samochod` zbudowaną z dwóch danych składowych: tablicy i zmiennej typu `float` oraz jednej funkcji składowej.

Choć potocznie mówimy "definiowanie klasy", to z punktu widzenia kompilatora mamy do czynienia z definicją odroczoną klasy.

> Definicja klasy jest definicją odroczoną, bo kompilator czytając definicję klasy nie generuje żadnego kodu, a jedynie uczy się jak klasa wygląda.

Kod generowany jest jedynie przy:

- kompilowaniu funkcji składowych klasy zdefiniowanych poza klasą (powiemy o tym później)
- definiowaniu składników statycznych klasy (powiemy o tym później)
- definiowaniu zmiennych (tworzeniu obiektów) wcześniej zdefiniowanej klasy.

Tworzenie obiektów

Na podstawie definicji klasy można stworzyć konkretny **obiekt** zdefiniowanej klasy.

Na przykład deklaracja

```
samochod kn32344;
```

tworzy obiekt zdefiniowanej wyżej klasy `samochod` i nadaje mu nazwę `kn32344`.

> - Definicja klasy jest tylko receptą, przepisem na nowy typ danych i sama od siebie nie tworzy żadnego obiektu.
> - Obiekty klasy tworzymy deklarując zmienne zdefiniowanej wcześniej klasy.
> - Na podstawie jednej definicji klasy można powołać do życia dowolnie wiele różnych obiektów (zadeklarować dowolnie wiele zmiennych danej klasy).

Na przykład deklaracje

```
samochod czerwony,niebieski;
samochod kr23456,wa11111;
```

tworzą łącznie cztery obiekty klasy `samochod`.

Dla każdego obiektu kompilator rezerwuje oddzielny fragment pamięci do przechowywania danych składowych tego obiektu tak, by każdemu z obiektów można było przypisywać inne wartości tych danych.

Odnoszenie się do elementów składowych

Poza definicją klasy lub funkcji składowej klasy do elementów składowych obiektu (tak danych jak i funkcji) odnosimy się przy użyciu **notacji kropkowej**.

```
czerwony.przebieg=23000;
kr23456.paliwo=20.5;
czerwony.jedz(120);
```

Jeśli dysponujemy wskaźnikiem do obiektu, stosujemy **notację strzałkową**

```
samochod *wskaznik_do_samochodu=&czerwony;
wskaznik_do_samochodu->paliwo=17.5;
wskaznik_do_samochodu->jedz(325);
```

Enkapsulacja

Korzystanie z klas umożliwia programiście gromadzenie danych wokół obiektów, do których się one odnoszą, dzięki czemu program zyskuje na czytelności.

Często warto pójść o krok dalej i uniemożliwić funkcjom spoza klasy dostęp do pewnych składowych klasy.

Pozostając przy przykładzie z samochodem można powiedzieć, że, przykładowo, kierownica jest elementem składowym samochodu, do którego na pewno powinien mieć dostęp jego użytkownik. Natomiast z pewnością lepiej jest, jeśli przeciętny użytkownik nie grzebie w skrzyni biegów.

Podobnie jest w przypadku klasy. Mogą być w niej zmienne składowe, do których dostęp powinny mieć tylko funkcje składowe, aby zapewnić, że nikt niezorientowany przez pomyłkę

lub umyślnie nie zmieni wartości tych zmiennych w sposób niedopuszczalny.

Możliwość chronienia składników klasy przed niewłaściwym użyciem jest drugą cechą charakterystyczną programowania obiektowego, tak zwaną **enkapsulacją**.

Typy ochrony danych

W C++ mamy do dyspozycji następujące typy ochrony danych:

`private`
: jest dostępny tylko dla funkcji składowych danej klasy. Jeżeli zależy nam, by nikt niepowołany nie grzebał w danym składniku, to powinien być on zadeklarowany właśnie jako prywatny.

`protected` (dla zorientowanych)
: jest dostępny tak, jak składnik `private`, ale dodatkowo jest jeszcze dostępny dla klas wywodzących się od tej klasy. Ten sposób ochrony stanie się zrozumiały po omówieniu dziedziczenia.

`public`
: jest dostępny bez ograniczeń. Zwykle składnikami takimi są jakieś wybrane funkcje składowe. To za ich pomocą dokonuje się z zewnątrz operacji na danych prywatnych.

Na przykład

```
class samochod {
 private:
   char model[20];
   float przebieg,paliwo;
 public:
   void jedz(float ile_kilometrow){
     przebieg=przebieg+ile_kilometrow;
   };
};
```

- Najczęściej dobrze zaprojektowana klasa daje dostęp bez ograniczeń tylko do wybranych funkcji składowych
- Klasy zdefiniowane przy użyciu `struct` i `class` różnią się tym, że w klasach zdefiniowanych przy użyciu `struct` domyślnym typem ochrony jest `public`, a w klasach zdefiniowanych przy użyciu `class` domyślnym typem ochrony jest `private`.

- W praktyce `struct` używane jest rzadko

Składniki klasy

- Składnikami klasy mogą być dane oraz funkcje.
- Dana składowa klasy może być
  - zmienną typu podstawowego (wbudowanego)
  - tablicą zmiennych typu podstawowego
  - zmienną (obiektem) wcześniej zdefiniowanej klasy
  - tablicą obiektów
- Funkcja składowa klasy może być w klasie zdefiniowana lub tylko zadeklarowana.
- Jeśli funkcja składowa jest tylko zadeklarowana, to jej definicja musi być umieszczona poza definicją klasy.

Funkcje składowe często określa się krótko mianem **metody**.

Na przykład w definicji

```
class samochod {
  ...
  void tankuj(float ile_litrow){
    paliwo+=ile_litrow;
  }
  void jedz(float ile_kilometrow);
  ...
}
```

funkcję `tankuj` zdefiniowano, a funkcję `jedz` jedynie zadeklarowano.

W takim przypadku definicja tej funkcji umieszczona poza klasą wygląda na przykład tak:

```
void samochod::jedz(float ile_kilometrow){
  float ile_przejedziemy=min(ile_kilometrow,paliwo*10);
  przebieg+=ile_przejedziemy;
  paliwo-=ile_przejedziemy/10;
}
```

- Podwójny dwukropek to **operator zakresu**.
- Nazwa klasy przed operatorem zakresu informuje kompilator, dla której klasy zdefiniowano funkcję składową.
- Jest to konieczne, bo nic nie stoi na przeszkodzie, by różne klasy miały funkcje składowe o tych samych nazwach.
- N.p. funkcja `tankuj` może pojawić się w definicji klasy `samolot`.

Funkcje zdefiniowane wewnątrz definicji klasy kompilator traktuje jak funkcje `inline`. Oznacza to, że kod dla takich funkcji generowany jest dopiero gdy są one wołane.

W praktyce w definicji klasy umieszczamy definicje funkcji składowych tylko wtedy gdy są to bardzo proste i krótkie definicje

Inicjalizacja zmiennych w klasie

Jeśli chcemy chronić dostęp do danych występujących w definicji klasy, możemy do nadawania im wartości używać specjalnych funkcji składowych.

```
   void samochod::inicjalizuj(char *jakis_model, float jakis_przebieg,
float iles_paliwa){
     strcpy(model,jakis_model);
     przebieg=jakis_przebieg;
     paliwo=iles_paliwa;
   }
   ...
   samochod wwa1123;
   wwa1123.inicjalizuj("Toyota Avensis",1200,20.7);
```

Metodę tę stosujemy normalnie tylko w celu umożliwienia innym funkcjom (kontrolowanej) zmiany wartości składników chronionych.

Konstruktory

W przypadku inicjalizacji korzystamy ze specjalnego mechanizmu oferowanego przez język C++

```
   samochod::samochod(char *jakis_model, float jakis_przebieg, float
iles_paliwa){
     strcpy(model,jakis_model);
     przebieg=jakis_przebieg;
     paliwo=iles_paliwa;
   }
   ...
   samochod wwa1123=samochod("Toyota Avensis",1200,20.7);
   samochod kwd1215("Honda Civic",44000,39.2);
```

- **Konstruktory** to specjalne funkcje składowe, które nazywają się tak samo jak klasa.
- Konstruktory są mechanizmem łączącym kreowanie obiektów z ich inicjalizacją.

- Jeśli klasa nie definiuje żadnego konstruktora, kompilator sam tworzy konstruktor postaci

```
nazwa_klasy()
```

- Jest to tak zwany **konstruktor domniemany**.

Lista inicjalizacyjna konstruktora

By uniknąć niepotrzebnego tworzenia obiektów chwilowych przy inicjalizacji obiektu używa się **list inicjalizacyjnych konstruktora**.

```
   samochod::samochod(char *jakis_model, float jakis_przebieg, float
iles_paliwa):
     przebieg(jakis_przebieg),paliwo(iles_paliwa){
     strcpy(model,jakis_model);
   }
```

Jeśli elementem składowym klasy jest obiekt jakiejś innej klasy, jego inicjalizację też najlepiej wykonać na liście inicjalizacyjnej.

Równoczesne zdefiniowanie klasy i utworzenie obiektów

Choć rzadko tak się robi, można równocześnie zdefiniować klasę i utworzyć jakieś jej obiekty.

```
struct Para{
  int lewy,prawy;
} pierwszaPara, drugaPara;
```

Wskaźnik `this`

- Funkcje składowe, w przeciwieństwie do danych składowych, nie są przechowywane oddzielnie dla każdego obiektu, ale tylko w jednym, wspólnym dla wszystkich klas egzemplarzu.
- Mimo to każda funkcja składowa wie na rzecz jakiego obiektu działa i jak dotrzeć do danych składowych tego obiektu.

- Jest to możliwe, ponieważ przy wywołaniu funkcji składowej na rzecz pewnego obiektu jest do niej przesyłany w sposób niejawny wskaźnik do tego obiektu.

Wskaźnik pokazujący na rzecz jakiego obiektu została wywołana funkcja składowa klasy dostępny jest poprzez słowo kluczowe `this`.

Unie

- Czasami dla zaoszczędzenia pamięci przydaje się obiekt, którego typ może się zmieniać w trakcie wykonywania programu.
- Do przechowywania tego typu obiektów w języku C++ służą **unie**.
- Składnia i użycie unii jest zbliżone do klas

```
union U{
  int i;
  float f;
}
U u;
u.f=2.78;
```

- W przeciwieństwie do klas, wszystkie dane składowe obiektu unii zajmują to samo miejsce a pamięci.
- Obiekt unii zajmuje tyle miejsca w pamięci co największy z jej składników
- W danym momencie w obiekcie unii może być przechowywana tylko jedna dana składowa
- Unie mogą mieć funkcje składowe, w tym konstruktory i destruktory
- Unie mogą chronić dostęp do składników słowami `public` i `private`
- Dla zorientowanych: składnikiem unii nie może zostać obiekt klasy, w której zdefiniowano konstruktor, destruktor, lub operator przypisania.
- Dla zorientowanych: w przypadku unii nie można stosować mechanizmu dziedziczenia

Unie anonimow

- Można zdefiniować dokładnie jeden egzemplarz unii.

```
union{
  int i;
  float f;
}
```

- Nie dajemy mu wtedy nazwy, ani nie nazywamy samej unii.
- O takiej unii mówimy, że jest **anonimowa**.
- Dostęp do składników unii anonimowej jest bezpośrednio poprzez ich nazwy.
- Unia anonimowa jest często składnikiem klasy.
- W takim przypadku dostęp do jej składników jest poprzez notację kropkową, w której przed kropką jest nazwa obiektu klasy, a po kropce nazwa składnika unii.