Programowanie 1

Sławomir Pluciński (splucinski@pjwstk.edu.pl)

Struktury danych w C++

definicja: złożony typ danych pozwalający na przechowywanie atrybutów o różnych typach w ramach jednej opisanej struktury.

Do utworzenia złożonego typu należy skorzystać ze słowa kluczowego **struct,** a następnie w nawiasach klamrowych zdefiniować nazwy pól i ich typy, które nie muszą być jednorodne.

```
struct Samochod{
    string marka;
    string model;
    int ilosc_biegow;
    double moc_silnika;
};
```

Gdy mamy tak stworzona strukturę, możemy teraz tworzyć zmienne na jej podstawie. Nie będzie się to niczym różniło od tego w jaki sposób tworzyliśmy dotychczas zmienne, jedynie jako typ będziemy w wskazywać ten utworzony przez nas. Alternatywnie można tworzyć zmienne od razu po definicji struktury.

```
struct Samochod{
    string marka;
    string model;
    int ilosc_biegow;
    double moc_silnika;
}fiat, mazda;
```

```
struct Samochod{
    string marka;
    string model;
    int ilosc_biegow;
    double moc_silnika;
};


int main() {
    Samochod fiat;

    return 0;
}
```

Odwołanie się do konkretnych wartości struktury będzie odbywać się z wykorzystaniem łącznika kropki. Najpierw należy wskazać nazwę struktury, a następie po kropce nazwę atrybutu do którego chcemy się odwołać.

```
struct Samochod{
    string marka;
    string model;
    int ilosc_biegow;
    double moc_silnika;
};


int main() {
    Samochod fiat;

    cin >> fiat.marka;
    cout << fiat.marka;

    return 0;
}
```

Aby nadać wartości dla poszczególnych atrybutów można odwoływać się po kolei do każdego pola osobno i przypisać wartość ale również można przypisać wartości inicjalne dla struktury w następujące sposoby:

```
struct Samochod{
    string marka;
    string model;
    int ilosc_biegow;
    double moc_silnika;
};


int main() {
    Samochod fiat = {"fiat", "126p", 4, 650.0};

    cout << fiat.marka;

    return 0;
}
```

```
struct Samochod{
    string marka;
    string model;
    int ilosc_biegow;
    double moc_silnika;
} fiat = {"fiat", "126p", 4, 650.0};


int main() {
    cout << fiat.marka;

    return 0;
}
```

Zasięg struktury może być różny w zależności gdzie zostanie zdefiniowana. Jeżeli struktura będzie znajdowała się nad metodą main() wtedy będziemy mówić o strukturze globalnej i będzie ona dostępna dla całej aplikacji. Jednak można też struktury tworzyć w ramach metod, czy to main(), czy to własnej metody i w konsekwencji dostęp do tej struktury będzie ograniczony.

```
struct Samochod{
    string marka;
    string model;
    int ilosc_biegow;
    double moc_silnika;
};


int main() {
    Samochod fiat;
    cout << fiat.marka;

    return 0;
}
```

```
void tworze_samochod(string nazwa);

int main() {
    tworze_samochod("fiat");
    return 0;
}

void tworze_samochod(string nazwa){
    struct Samochod{
        string marka;
        string model;
        int ilosc_biegow;
        double moc_silnika;
    };
    Samochod nowy;
    nowy.marka = nazwa;
    cout << nowy.marka;
}
```

Typy wyliczeniowe

definicja: typ danych wykorzystywany do przechowywana zbioru stałych wartości. W swojej budowie przypomina strukturę, a w zachowaniu stałą **const**. Aby utworzyć typ wyliczeniowy należy skorzystać ze słowa kluczowego **enum.**

```
enum dzien{
    poniedzialek,
    wtorek,
    sroda,
    czwartek,
    piatek,
    sobota,
    niedziela
};
```

Wartości dla wskazanych poszczególnych atrybutów możemy nadpisać. Domyślnie przyjmują one kolejne liczby zaczynając od 0.

```
enum dzien {
    poniedzialek = 1,
    wtorek =2,
    sroda = 3,
    czwartek = 4,
    piatek = 5,
    sobota = 6,
    niedziela = 7,
    brak = -1
};
```

Odwołanie się do konkretnego atrybutu odbywa się w następujący sposób:

```
int main() {

enum dzien {
    poniedzialek = 1,
    wtorek =2,
    sroda = 3,
    czwartek = 4,
    piatek = 5,
    sobota = 6,
    niedziela = 7,
    brak = -1
};

dzien dzien_tygodnia = poniedzialek;
cout << dzien_tygodnia;

int numer_dnia = piatek;
cout << numer_dnia;

}
```

Programowanie zorientowane obiektowo w C++

definicja: Klasa jest typem złożonym zawierającym definicję atrybutów i dostępnych metod możliwych do wywołania w kontekście klasy. Do utworzenia klasy należy wykorzystać słowo kluczowe **class,** a następnie wskazać nazwę klasy**.**

```
class nazwa_klasy {
    //atrybuty
    //funkcje
};
```

```
class Pomieszczenie {
    public:
        double dlugosc;
        double szerokosc;

        double oblicz_powierzchnie(){
            return dlugosc * szerokosc;
        }
};
```

Po zdefiniowany klasy staje się ona dostępna jako typ danych, który możemy przypisać do zmiennej. W ten sposób utworzony zostanie obiekt danej klasy.

```
class Pomieszczenie {
    public:
        double dlugosc;
        double szerokosc;

        double oblicz_powierzchnie(){
            return dlugosc * szerokosc;
        }
};

int main() {
    Pomieszczenie sala_wykladowa;
    Pomieszczenie sala_216;
    Pomieszczenie sala_110;
}
```

Dostęp do zmiennych, czy funkcji realizowany jest przez wykorzystanie operatora kropki. Najpierw należy wskazać obiekt w ramach którego chcemy się poruszać, a następnie po kropce pole do którego chcemy się dostać lub nazwę metody, którą chcemy wywołać.

```
class Pomieszczenie {
    public:
        double dlugosc;
        double szerokosc;

        double oblicz_powierzchnie(){
            return dlugosc * szerokosc;
        }
};

int main() {
    Pomieszczenie sala_wykladowa;

    sala_wykladowa.dlugosc = 10;
    sala_wykladowa.szerokosc = 30;
    cout << sala_wykladowa.oblicz_powierzchnie();

}
```

Zmienne i funkcje w ramach klasy mogą mieć ograniczoną widoczność. Nie wszystkie muszą być dostępne jak tylko utworzymy obiekt danej klasy. Dostępem do zawartości klasy będą sterować modyfikatory dostępu public / private / protected.

```
class Pomieszczenie {
    private:
        double dlugosc;
        double szerokosc;

    public:
        double oblicz_powierzchnie(){
            return dlugosc * szerokosc;
        }
};

int main() {
    Pomieszczenie sala_wykladowa;
    cout << sala_wykladowa.oblicz_powierzchnie();
}
```

W przypadku ograniczenia dostępu do atrybutów klasy pojawia się problem z ustawieniem dla nich wartości. Nie mając dostępu do zmiennej nie jesteśmy w stanie wskazać jakie powinny być dla nich wartości. Jako rozwiązanie tego problemu można udostępnić funkcję, która umożliwi ustawianie wartości zmiennej (setter).

```
class Pomieszczenie {
    private:
        double dlugosc;
        double szerokosc;

    public:

        void init(double dlug, double szer){
            dlugosc = dlug;
            szerokosc = szer;
        }

        double oblicz_powierzchnie(){
            return dlugosc * szerokosc;
        }
};

int main() {
    Pomieszczenie sala_wykladowa;
    sala_wykladowa.init(10, 30);
    cout << sala_wykladowa.oblicz_powierzchnie();
}
```

```
class Pomieszczenie {
    private:
        double dlugosc;
        double szerokosc;

    public:
        void setDlugosc(double dlug){
            dlugosc = dlug;
        }
        void setSzerokosc(double szer){
            szerokosc = szer;
        }
        double oblicz_powierzchnie(){
            return dlugosc * szerokosc;
        }
};

int main() {
    Pomieszczenie sala_wykladowa;
    sala_wykladowa.setDlugosc(10);
    sala_wykladowa.setSzerokosc(30);
    cout << sala_wykladowa.oblicz_powierzchnie();
}
```

Alternatywnie możemy skorzystać ze specjalnej metody klasy, którą będziemy nazywać **konstruktorem**. Zadaniem tej metody jest umożliwienie tworzenie obiektów danej klasy. W sytuacji kiedy nie utworzymy żadnego konstruktora w klasie to automatycznie tworzony jest tak zwany konstruktor domyślny.

```
class Pomieszczenie {
    private:
        double dlugosc;
        double szerokosc;
    public:
        Pomieszczenie(){

        }
};
```

Z uwagi na to, że konstruktor jest traktowany jak metoda, możemy w nim wywoływać dowolny kod i mieć dostęp do zmiennych, które są w zasięgu widoczności dla konstruktora. Możemy przez konstruktor nie tylko tworzyć obiekty ale również nadawać im wartości inicjalne.

```
class Pomieszczenie {
    private:
        double dlugosc;
        double szerokosc;

    public:
        Pomieszczenie(double dlug, double szer){
            dlugosc = dlug;
            szerokosc = szer;
        }

        double oblicz_powierzchnie(){
            return dlugosc * szerokosc;
        }
};

int main() {
    Pomieszczenie sala_wykladowa(1.0,1.0);
    cout << sala_wykladowa.oblicz_powierzchnie();
}
```

```
class Pomieszczenie {
    private:
        double dlugosc;
        double szerokosc;

    public:
        Pomieszczenie(double dlug, double szer){
            dlugosc = dlug;
            szerokosc = szer;
        }

        double oblicz_powierzchnie(){
            return dlugosc * szerokosc;
        }
};

int main() {
    Pomieszczenie sala_wykladowa;
    cout << sala_wykladowa.oblicz_powierzchnie();
}
```

Każdy obiekt ma swój czas trwania i po jego utworzeniu musi przyjść taki moment, kiedy zostanie zniszczony. Do tego zadania wykorzystuje się destruktory, które też możemy implementować. Jest on wywołany automatycznie przy niszczeniu obiektu. W ramach klasy może wystąpić tylko jeden.

```
class Pomieszczenie {
    private:
        double dlugosc;
        double szerokosc;

    public:
        Pomieszczenie(double dlug, double szer){
            dlugosc = dlug;
            szerokosc = szer;
        }

        ~Pomieszczenie(){
            cout << "To tyle na dziś - Dziękuję"
        }
};

int main() {
    Pomieszczenie sala_wykladowa(5.0, 10.0);
    Pomieszczenie sala110 = sala_wykladowa;

}
```