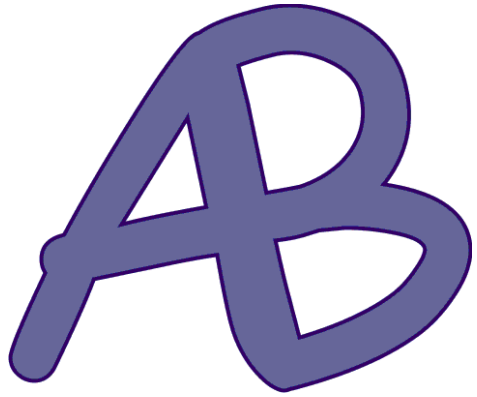


Programowanie obiektowe

Klasy - wstęp

dr Artur Bartoszewski

Klasy

- Programowanie obiektowe (ang. object-oriented programming, OOP) – paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących stan (czyli dane, nazywane najczęściej polami) i zachowanie (czyli procedury, tu: metody). Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

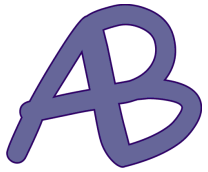
- Podejście to różni się od tradycyjnego programowania proceduralnego, gdzie dane i procedury nie są ze sobą bezpośrednio związane. Programowanie obiektowe ma ułatwić pisanie, konserwację i wielokrotne użycie programów lub ich fragmentów.

Źródło: Programowanie obiektowe – Wikipedia, wolna encyklopedia

Klasy

Główne cechy programowania obiektowego:

1. Hermetyzacja (Encapsulation) – ukrywanie szczegółów implementacji i udostępnianie tylko tych elementów, które są niezbędne do interakcji z obiektem.
2. Dziedziczenie (Inheritance) – możliwość tworzenia nowych klas na podstawie istniejących, co pozwala na ponowne wykorzystanie kodu.
3. Polimorfizm (Polymorphism) – możliwość definiowania metod o tej samej nazwie, ale o różnym działaniu w zależności od kontekstu.
4. Abstrakcja (Abstraction) – tworzenie modeli rzeczywistości poprzez definiowanie klas zawierających tylko istotne informacje.



Klasa i obiekt

Klasa to szablon,
który służy do tworzenia **obiektów.**

Klasy

- Klasa to, najprościej mówiąc, złożony typ danych zawierający zbiór informacji (danych składowych) oraz sposób ich zachowania.
- Klasę można uznać za model jakiegoś rzeczywistego obiektu.

```
class Nazwa_Klasy
{
        // ciało klasy - w tym miejscu
        // piszemy definicje typów,
        // zmienne  i funkcje jakie mają należeć
        // do klasy.
};
        //uwaga na średnik!
```

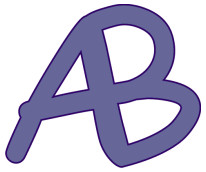
Klasy i egzemplarze (obiekty) klasy

- Klasa to najprościej mówiąc projekt lub schemat. Aby jej używać należy stworzyć egzemplarz (obiekt) danej klasy.
- Obiektem nazywamy egzemplarz klasy. Tworzymy go tak jak zmienną (w istocie, to jest zmienna – rozbudowana wersją poznanej już zmiennej struct).

```
class TwojaKlasa
{
};
int main()
{
    TwojaKlasa obiektKlasy;
    return 0;
}
```

TwojaKlasa to tylko opis – jeszcze nie istniej żaden obiekt tej klasy

obiektKlasy to zmienna (obiekt typu **TwojaKlasa**) z której można korzystać



Składniki klasy

- ✓ pola / atrybuty
- ✓ metody

Składniki klasy (pola i metody)

Na klasę składają się **zmienne** przechowujące dane oraz **funkcje** które na tych danych operują..

- ✓ zmienne w klasie nazywamy polami lub atrybutami
- ✓ funkcje w klasie nazywamy metodami
- ✓ funkcje i zmienne w klasie nazywamy ogólnie składnikami klasy

Składniki klasy (pola i metody)

```
class osoba
 {
    public:
      string imie;          // pola (zmienne)
      string nazwisko;
      int  wiek;
} ;
```

Każdy obiekt danej klasy (jej instancja) posiada **własny zestaw atrybutów** (chyba że są statyczne – wtedy należą do całej klasy, a nie do konkretnego obiektu).

Przykład:

```
class Osoba
{
public:
    string imie;
    string nazwisko;
    int wiek;
};

int main()
{
    Osoba pracownik;
    pracownik.imie = "Jan";
    pracownik.nazwisko = "Kowalski";
    pracownik.wiek = 30;

    cout << "Imie: " << pracownik.imie << endl
         << "Nazwisko: " << pracownik.nazwisko << endl
         << "Wiek: " << pracownik.wiek << endl;
    return 0;
}
```

Klasa – szablon dla obiektów

Obiekt – instancja klasy – konkretna zmienna przechowująca własne dane

Składniki klasy (pola i metody)

Aby odwołać się do składników obiektu możemy posłużyć się jedną z poniższych notacji:

Dla obiektów utworzonych statycznie:

```
obiekt.składnik;
```

Dla obiektów utworzonych dynamicznie (za pomocą **new**):

```
wskaźnik -> składnik;
```

Dla obiektów do których utworzyliśmy referencję:

```
referencja.składnik;
```

Składniki klasy (pola i metody)

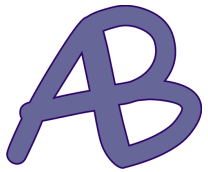
Przykład składni:

```
osoba pracownik1;

pracownik1.wiek = 40;          //nazwa obiektu

osoba *wsk = &pracownik1;      //wskaźnik
cout << wsk->wiek;

osoba &robol = pracownik1;     //referencja
cout << robol.wiek
```



Enkapsulacja (hermetyzacja)

- Enkapsulacja zapewnia, że program, ani inny obiekt nie może zmieniać stanu wewnętrznego obiektów w nieoczekiwany sposób.
- Tylko własne metody obiektu są uprawnione do zmiany jego stanu.
- Każdy typ obiektu prezentuje innym obiektom swój interfejs, który określa dopuszczalne metody współpracy.

Programowanie obiektowe – Wikipedia, wolna encyklopedia

Enkapsulacja

Definiując dostęp do składników klasy używamy trzech słów kluczowych:

1. public – dostęp do składników klasy jest dozwolony wszędzie, nawet z poza ciała klasy,
2. private – dostęp do składników klasy jest zabroniony z poza ciała klasy (możliwy z funkcji zaprzyjaźnionej*),
3. protected – dostęp do składników klasy jest dozwolony tylko z ciała klasy (tak jak private) oraz w klasach pochodnych klasy bazowej.

Uwaga: wszystkie składniki są domyślnie prywatne czyli jeżeli nie podamy żadnego kwalifikatora dostępu wszystko wewnątrz klasy będzie prywatne.

* Funkcje zaprzyjaźnione poznany na kolejnym wykładzie

Enkapsulacja

```
class Osoba
{
    int Id;                  } pole prywatne (domyślnie)
public:
    string imie;             }
    string nazwisko;         } pola publiczne
protected:
    int wiek;                } pole chronione
private:
    string PESEL;            } pole prywatne
};
```

Enkapsulacja

Enkapsulacja polega na ukrywaniu szczegółów implementacyjnych klasy przed programem, który tą klasę wykorzystuje.

Dobrze zaprojektowane klasy rozdzielają wewnętrzne mechanizmy (implementację) obiektu od metod służących jego obsłudze. Wtedy, programista wykorzystujący obiekt komunikuje się z nim przez jego API bez konieczności analizowania, co wykonywane jest pod spodem.

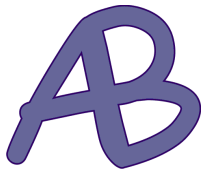
Enkapsulacja sprawia, że poszczególne klasy mogą być testowane i rozwijane w izolacji. Dzięki temu można pracować na nich bez ryzyka uszkodzenia innych elementów składowych programu.

Enkapsulacja

Klasa w C++ jest podobna do struktury (struct). Na tym etapie można zauważyć, że istnieją dwie różnice:

1. Inne słowo kluczowe,
2. Domyślny dostęp do składników:
   - w strukturze wszystkie składniki są publiczne,
   - w klasie wszystkie składniki są domyślnie prywatne.

Klasa ma jednak daleko większe możliwości, które poznany w trakcie kolejnych wykładów (dziedziczenie, polimorfizm itp.)



Metody
(funkcje składowe klas)

Metoda to funkcja zdefiniowana wewnątrz klasy, która opisuje zachowanie obiektów tej klasy.

- ✓ Metody działają na danych składowych klasy (pola/atrybuty).
- ✓ Metoda jest częścią interfejsu klasy – określa, jak można manipulować danymi obiektu i jakie operacje można na nim wykonać.

Metody

```
class Osoba
 {
    public:
       string imie;                  // pola (zmienne)
       string nazwisko;
       int wiek;

       int podaj_wiek()              // metoda (funkcja
       {
          // treść funkcji
       }
} ;
```

Metody

Definiowanie metod wewnątrz klasy

- Metody których ciało (pełna treść funkcji) opisane są wewnątrz definicji obiektu traktowane są jako metody inline.

- Oznacza to, że kod tej funkcji jest przez kompilator „wklejany” do każdego utworzonego obiektu tej klasy (pracujemy wtedy na tym samym segmencie pamięci, oszczędzając czas, jednak program zajmuje więcej pamięci).

- Ten sposób definiowania stosujemy dla metod krótkich (najczęściej składających się z jednego do trzech poleceń).

Metody

```
class osoba
{
    private:
      string imie;
      string nazwisko;
      int wiek;
    public:
    void setImie(string kto)
    {
        imie = kto;
    }
    void setNazwisko(string kto)
    {
        nazwisko = kto;
    }
    void setWiek(int ile)
    {
        wiek = ile;
    }
};
```

Należy pamięta, że metody odczytujące pola klasy muszą być publiczne (wywołujemy je z zewnątrz klasy)

Metody

```
int main()
{
    osoba pracownik1;
    string im, nazw;
    int w;
    cout<<"Podaj imie: ";          cin>>im;
    cout<<"Podaj nazwisko: ";      cin>>nazw;
    cout<<"Podaj wiek: ";          cin>>w;
    //pracownik1.imie = im;
    //powyższa operacje jest niedozwolna - pola są prywatne
    pracownik1.setImie(im);
    pracownik1.setNazwisko(nazw);
    pracownik1.setWiek(w);
    return 0;
```

Metody zapewniają dostęp do pól prywatnych.

Metody

Program z poprzedniego slajdu daje nam możliwość wstawania danych do pól obiektu (ogólniej ustawiania ich wartości) poprzez metody, ale zabrania ich odczytu.
Aby odczytać zawartość obiektu należy uzupełnić go o metody:

```
string getImie() {
    return imie;
}
string getNazwisko() {
    return nazwisko;
}
int getWiek() {
    return wiek;
}
```

Metody dodajemy do wnętrza (ciała) klasy koniecznie po modyfikatorze dostępu public:

```
cout << pracownik1.getImie()<<" "
     << pracownik1.getNazwisko()<<" "
     << pracownik1.getWiek();
```

W programie możemy ich użyć do odczytania zawartości pól prywatnych

Metody

Definiowanie metod poza klasą

W tym podejściu deklarujemy metodę w klasie, ale jej definicję umieszczamy na zewnątrz.

- ✓ Wewnątrz kasy umieszczamy nagłówek metody.
- ✓ Poza klasą (ale nie wewnątrz funkcji main) umieszczamy treść metody.
- ✓ Aby określić, że metoda należy do klasy posługujemy się operatorem przestrzeni nazw ::

```
void klasa::metoda( )
{ … }
```

Metody

Nagłówek metody umieszczamy wewnątrz klasy. Nagłówek nie ma ciała klasy (nawiasów klamrowych). Zamiast tego kończy się średnikiem

Pełną treść funkcji umieszczamy poza klasą należy pamiętać o zadeklarowaniu że funkcja talerzy w przestrzeni nazw klasy.

```
class osoba
{
private:
    string imie;
    string nazwisko;

public:
    void setImie(string im) { imie = im; }
    void setNazwisko(string nazw) { nazwisko = nazw; }
    string getImie() { return imie; }
    string getNazwisko() { return nazwisko; }
    void wizytowka();
};

void osoba::wizytowka()
{
    cout << endl
         << "--------------------------------" << endl
         << "Imie:\t\t" << imie << endl
         << "Nazwisko:\t" << nazwisko
         << "--------------------------------" << endl;
}
```

Metody

Definiowanie metod wewnątrz klasy

Zalety:

- ✓ Może zwiększyć wydajność, ponieważ eliminuje narzut związany z wywołaniem funkcji
- ✓ Łatwiejsza czytelność dla małych i prostych metod.
- ✓ Może poprawić optymalizację kodu przez kompilator

Wady:

- ✓ Może prowadzić do większego rozmiaru kodu binarnego, jeśli metoda jest często wywoływana w różnych miejscach (rozszerza kod zamiast stosować wywołanie funkcji).
- ✓ Zmiany w metodzie wymagają ponownej rekompilacji wszystkich plików, które dołączają nagłówek zawierający klasę.

Definiowanie metod poza klasą

Zalety:

- ✓ Lepsza organizacja kodu – definicje metod mogą znajdować się w osobnym pliku .cpp, co ułatwia zarządzanie dużymi projektami.
- ✓ Unikanie powielania kodu – metoda istnieje w jednym miejscu w pamięci zamiast być wielokrotnie wstawiana do kodu maszynowego.

Wady:

- ✓ Może mieć gorszą wydajność (każde wywołanie metody wiąże się z rzeczywistym wywołaniem funkcji w kodzie maszynowym).

Metody

Dzięki enkapsulacji pól obiektu, może on przechowywać dane w innej formie niż je przedstawia na zewnątrz.

Np.: w tej wersji metoda *osoba* pobiera **wiek**, ale wewnętrznie przechowuje **rok urodzenia,** dzięki czemu baza danych nie musi być co roku aktualizowana.

```
int main()
{
    osoba pracownik1;
    string im, nazw;
    int w;
    cout<<"Podaj imie: ";        cin>>im;
    cout<<"Podaj nazwisko: ";    cin>>nazw;
    cout<<"Podaj wiek: ";        cin>>w;
    //pracownik1.imie = im;
    //powyższa operacje jest niedozwolna - pola
    pracownik1.setImie(im);
    pracownik1.setNazwisko(nazw);
    //pracownik1.setWiek(w);
    pracownik1.podajWiek(w);

    cout << pracownik1.getImie()<<" "
         << pracownik1.getNazwisko()<<" "
         << pracownik1.getRokUrodzenia();
    return 0;
}
```

```
class osoba
{
    private:
      string imie;
      string nazwisko;
      int rokUrodzenia;
    public:
    void setImie(string kto) {
    void setNazwisko(string kto) {
    string getImie() {
    string getNazwisko() {
    int getRokUrodzenia() {
        return rokUrodzenia;
    }
     void podajWiek(int wiek);
};

void osoba::podajWiek(int wiek)
{
    time_t czas = time(NULL);
    tm * czaslokalny;
    czaslokalny=localtime(&czas);
    rokUrodzenia=(czaslokalny->tm_year+1900)-wiek;
}
```

Dobre praktyk programowania – metody dostępowe

Dobra praktyką jest tworzenie pól klas jako elementów prywatnych – niedostępnym z zewnątrz.
Do ich odczytywania i modyfikacji używamy wtedy metod publicznych o ustandaryzowanych nazwach.

Nazwy metod umożliwiających odczyt pól klasy nazywamy akcesorami (getterami). Ich nazwy rozpoczynamy zwyczajowo od przedrostka „get”.

Nazwy metod umożliwiających modyfikację pól klasy nazywamy mutatorami (setterami). Ich nazwy rozpoczynamy zwyczajowo od przedrostka „set”.

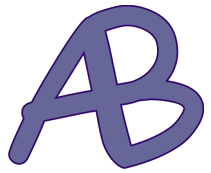
```
class Punkt
{
private:
    double x;
    double y;

public:
    //akcesory (gettery)
    double getX() { return x; }

    double getY() { return y; }

    //mutatory (settery)
    void setX(double px) { x = x; }

    void setY(double py) { y = y; }
};
```



Statyczne oraz dynamiczne tworzenie obiektów

W C++ obiekty mogą być tworzone na dwa główne sposoby:

- statycznie,
- dynamicznie.

Różnica między tymi metodami polega na czasie tworzenia obiektu, zarządzaniu pamięcią i jego zasięgu

Statyczne i dynamiczne tworzenie obiektów

Statyczne tworzenie obiektów

Obiekty tworzone statycznie są tworzone na stosie (stack) w momencie deklaracji zmiennej. Oznacza to, że obiekt jest tworzony w czasie kompilacji i jest automatycznie niszczony, gdy wychodzimy z zakresu, w którym został zadeklarowany.

Cechy obiektów statycznych:

- ✓ Obiekt ma zasięg lokalny, czyli jest dostępny tylko w funkcji lub bloku, w którym został zadeklarowany.
- ✓ Czas życia obiektu jest zarządzany automatycznie — trwa od momentu utworzenia do momentu zakończenia funkcji lub bloku, w którym został zadeklarowany.
- ✓ Tworzenie obiektów statycznych jest szybkie, ponieważ obiekt jest przechowywany na stosie, a jego pamięć jest automatycznie zarządzana.

Statyczne i dynamiczne tworzenie obiektów

Statyczne tworzenie obiektów – przykład:

```
class Osoba
{
public:
    string imie;
    string nazwisko;
};

int main()
{
    Osoba pracownik;
    pracownik.imie = "Jan";
    pracownik.nazwisko = "Kowalski";

    cout << "Imie: " << pracownik.imie << endl
         << "Nazwisko: " << pracownik.nazwisko << endl;
    return 0;
}
```

Statyczne i dynamiczne tworzenie obiektów

Dynamiczne tworzenie obiektów

Obiekty tworzone dynamicznie są tworzone w dynamicznej pamięci (heap) za pomocą operatora new. Obiekt taki nie jest automatycznie usuwany po zakończeniu funkcji, co oznacza, że musisz samodzielnie zarządzać jego pamięcią, używając operatora delete.

Cechy obiektów dynamicznych:

- ✓ Obiekt dynamiczny może być dostępny z dowolnego miejsca w programie, dopóki nie zostanie usunięty.
- ✓ Czas życia obiektu jest niezależny od zasięgu, w którym został zadeklarowany. Obiekt będzie istniał, dopóki nie zostanie usunięty za pomocą operatora delete.
- ✓ Tworzenie obiektów dynamicznych jest wolniejsze i bardziej zasobożerne niż tworzenie obiektów statycznych, ponieważ wymaga zarządzania pamięcią na stercie.

Statyczne i dynamiczne tworzenie obiektów

Dynamiczne tworzenie obiektów - przykład

```
class Osoba
{
public:
    string imie;
    string nazwisko;
};

int main()
{
    Osoba * pracownik = new Osoba;
    pracownik->imie = "Jan";
    pracownik->nazwisko = "Kowalski";
    cout << "Imie: " << pracownik->imie << endl
         << "Nazwisko: " << pracownik->nazwisko << endl;
    delete pracownik;
    return 0;
}
```

Statyczne i dynamiczne tworzenie obiektów

	Statyczne	Dynamiczne
Miejsce przechowywania	Stos (stack)	Sterta (heap)
Czas życia	Automatycznie zarządzany (żyje do końca zakresu)	Trwa, aż nie wywołasz delete
Zasięg	Lokalny (w obrębie funkcji/bloku)	Globalny (w obrębie programu, do usunięcia)
Zarządzanie pamięcią	Automatyczne	Ręczne (należy użyć delete)
Wydajność	Szybsze, ponieważ używa stosu	Wolniejsze, z powodu alokacji na stercie

Stos VS sterta

Stos (Stack)

Stos to obszar pamięci przeznaczony do alokacji automatycznej zmiennych lokalnych i parametrów funkcji.

Cechy stosu:

- ✓ Szybka alokacja i dealokacja – pamięć jest zwalniana automatycznie po zakończeniu zakresu zmiennej.
- ✓ Zarządzanie przez kompilator – nie musimy ręcznie zwalniać pamięci.
- ✓ Działa na zasadzie LIFO (Last In, First Out) – ostatnio zadeklarowane zmienne są usuwane jako pierwsze.
- ✓ Mały rozmiar – stos ma ograniczoną pojemność dlatego duże obiekty mogą prowadzić do przepełnienia stosu (Stack Overflow).

Stos VS sterta

Sterta (Heap)

Sterta to obszar pamięci przeznaczony do dynamicznej alokacji pamięci – gdy nie znamy z góry rozmiaru potrzebnej pamięci lub chcemy przechowywać obiekty przez dłuższy czas.

Cechy stosu:

- ✓ Pamięć musi być zarządzana ręcznie – programista musi samodzielnie alokować (new) i zwalniać (delete) pamięć.
- ✓ Dłuższy czas alokacji – sterta jest wolniejsza od stosu, ale daje większą elastyczność.
- ✓ Brak automatycznej dealokacji – nieuwolniona pamięć prowadzi do wycieków pamięci.
- ✓ Brak ograniczenia rozmiaru (w teorii) – ale zależy od dostępnej pamięci RAM.

Literatura:

W prezentacji wykorzystano przykłady i fragmenty:

- Gręboszj J. : ***Symfonia C++, Programowanie w języku C++ orientowane obiektowo***, Wydawnictwo Edition 2000.
- Jakubczyk K.: *Turbo Pascal i Borland C++ Przykłady*, Helion.

Warto zajrzeć także do:

- Sokół R. : ***Microsoft Visual Studio 2012 Programowanie w Ci C++,*** Helion.
- Kerninghan B. W., Ritchie D. M.: ***język ANSI C***, Wydawnictwo Naukowo Techniczne.

Dla bardziej zaawansowanych:

- Gręboszj J. : ***Pasja C++,*** Wydawnictwo Edition 2000.
- Meyers S.: ***język C++ bardziej efektywnie***, Wydawnictwo Naukowo Techniczne