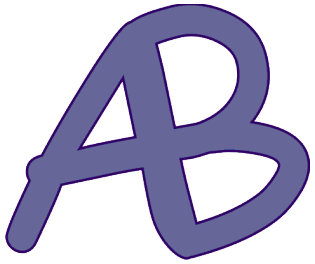


Wykład 2

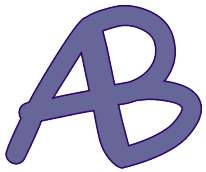
Klasy cz. 1

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



Klasa i obiekt

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

OBIEKT

OBIEKT

OBIEKT

OBIEKT

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!
```

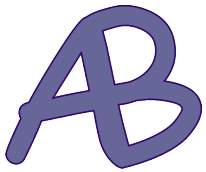
Klasy i egzemplarze (obiekty) klasy

- Klasa to projekt. Aby jej używać należy stworzyć egzemplarz (obiekt) danej klasy.
- Obiektem nazywamy egzemplarz klasy. Tworzymy go tak jak zmienną (w istocie, to jest zmienna).

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

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

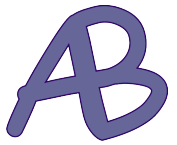
obiektKlasy to zmienna (obiekt typu **TwojaKlasa**) z której można korzystać



Składniki klasy

- pola
- metody

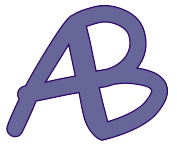
Składniki klasy (pola i metody)



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

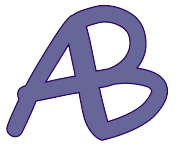
- zmienne w klasie nazywamy polami
- funkcje w klasie nazywamy metodami
- funkcje i zmienne w klasie nazywamy ogólnie składnikami klasy

Składniki klasy (pola i metody)



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

Przykład:



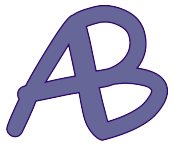
```
using namespace std;

class osoba
{
    public:
      string imie;
      string nazwisko;
      int wiek;
};

int main()
{
    osoba pracownik1;
    cout<<"Podaj imie: ";        cin>>pracownik1.imie;
    cout<<"Podaj nazwisko: ";    cin>>pracownik1.nazwisko;
    cout<<"Podaj wiek: ";        cin>>pracownik1.wiek;

    cout << pracownik1.imie<<" "
         << pracownik1.nazwisko<<" "
         << pracownik1.wiek;
    return 0;
}
```

Składniki klasy (pola i metody)



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

Dla obiektów utworzonych statycznie:

```
obiekt.składnik;
```

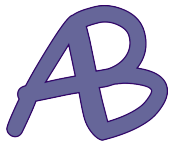
Dla obiektów utworzonych dynamicznie (za pomocą **new**):

```
wskaźnik -> składnik;
```

Dla obiektów do których utworzyliśmy referencję:

```
referencja.składnik;
```

Składniki klasy (pola i metody)



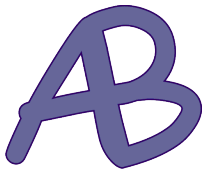
Przykład składni:

```
osoba pracownik1;

pracownik1.wiek = 40;          //nazwa obiektu

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

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

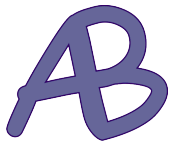


Enkapsulacja (hermetyzacja)

- 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 3 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 klasy zaprzyjaźnionej),
3. protected – dostęp do składników klasy jest dozwolony tylko z ciała klasy (tak jak private) oraz w klasach pochodnych klasy bazowej.

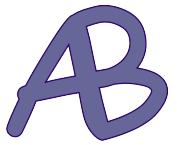
Uwaga: wszystkie składniki są domyślnie prywatne, chyba że programista będzie chciał inaczej.

Enkapsulacja

```
class osoba {

    int id;                  //pole prywatne
public:
    string imie;             //pole publiczne
    string nazwisko;         //pole publiczne
    void WypiszImie();       //metoda publiczna
protected:
    int wiek;                //pole chronione
    int wzrost;              //pole chronione
private:
    int pesel;               //pole prywatne
    int nr_dowodu;           //pole prywatne
    void WypiszPesel();      //metoda prywatna
};
```

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.

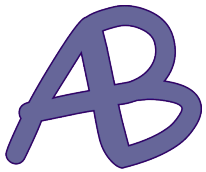
Enkapsulacja sprawia, że poszczególne klasy mogą być testowane i rozwijane w izolacji. Dzięki temu można pracować na nich bez ryzyka uszkodzenia innych elementów składowych programu.

Enkapsulacja

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

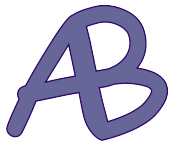
1. Inne słowo kluczowe,
2. Domyślny dostęp do składników:
   - w strukturze wszystkie składniki są publiczne,
   - w klasie wszystkie składniki są domyślnie prywatne.

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



Metody
(funkcje składowe klas)

Metody



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

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

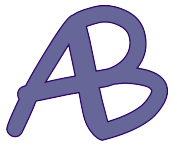
Metody

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

- Oznacza to, że kod tej funkcji jest „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

Bardziej rozbudowane metody nie powinny być tworzone jako metody inline (powielane przy każdym egzemplarzu klasy).

Jeżeli metoda nie powinna być traktowana jako funkcja inline należy zdefiniować ja poza klasą.

- 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

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);
};

int main()
{

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**”.

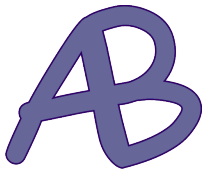
```
class Punkt
{
private:
    double x;
    double y;

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; }
};
```



Na marginesie...
Zegar czasu rzeczywistego

Objaśnienie funkcji **time ( )** i **localtime ( )** użytych w poprzednim przykładzie

Zagadnienie niezwiązane bezpośrednio z tematem wykładu, ale wykorzystane w przykładach.

Zegar

- Zegar czasu rzeczywistego funkcja time( ) pozwala pobrać aktualny czas zegara czasu rzeczywistego podanego jako ilość sekund która upłynęła od 1 stycznia 1970r.
- Wynik zapisany jest do zmiennej typu time_t

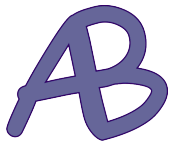
```
#include <ctime>
#include <iostream>
using namespace std;

int main()
{
    time_t aktualnyCzas;

    aktualnyCzas = time(NULL);
    //time(&aktualnyCzas);
    cout << aktualnyCzas << " sekund uplynelo od 00:00:00, 01.01.1970r";

    return 0;
}
```

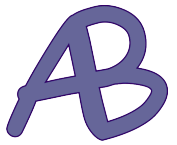
Zegar



Struktura **tm** przechowuje składowe daty i czasu w postaci liczb.

```
#include <ctime>
struct tm
{
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

Zegar



tm_sec	Sekundy [0..59]
tm_min	Minuty [0..59]
tm_hour	Godziny [0..23]
tm_mday	Dzień miesiąca [1..31]
tm_mon	Miesiąc [0..11]
tm_year	Obecny rok. Lata zaczynają się liczyć od roku 1900, czyli: wartość 0 = 1900 rok.
tm_wday	Dzień tygodnia. Zakres [0..6]. Znaczenie poszczególnych wartości: 0 = Niedziela 1 = Poniedziałek 2 = Wtorek 3 = Środa 4 = Czwartek 5 = Piątek 6 = Sobota
tm_yday	Dzień roku. Zakres [0..365].
tm_isdst	Letnie/zimowe przesunięcie czasowe. Jeśli wartość jest większa od 0 to przesunięcie czasowe jest 'aktywne'. Jeśli wartość mniejsza od 0 to informacja jest niedostępna.

Zegar

- Funkcja localtime( ) zamienia czas pobrany z zegara na czytelną postać.
- Wynik zapisywany jest jako wskaźnik do predefiniowanej struktury tm, której polami są lata, miesiące, dni, godziny, minuty i sekundy.

```
int main()
{
    time_t aktualnyCzas;
    aktualnyCzas = time(NULL);
    //time(&aktualnyCzas);
    tm *czasLokalny = localtime(&aktualnyCzas);
    cout <<czasLokalny->tm_hour<<" "
         <<czasLokalny->tm_min<<" "
         <<czasLokalny->tm_sec<<endl;
    cout <<czasLokalny->tm_year+1900<<" "
         <<czasLokalny->tm_mon+1<<" "
         <<czasLokalny->tm_mday<<endl;
    return 0;
```

Literatura:

W prezentacji wykorzystano przykłady i fragmenty:

- Gręboszl 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:

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