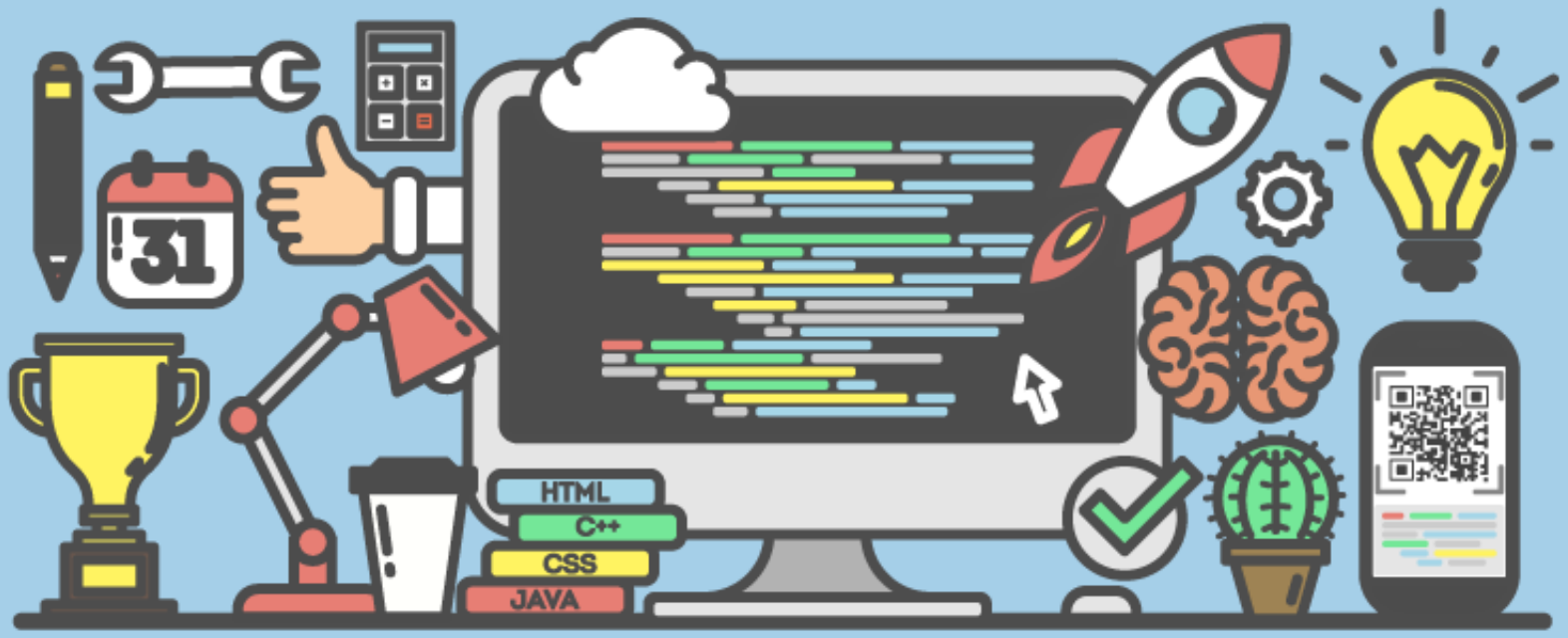
# Programowanie w C++

## Kurs (średnio)zaawansowany

### Zajęcia numer 17

**Rafał Berdyga**
**rberdyga@gmail.com**




# Plan na dzisiaj

- Przykładowe zadania związane z pisaniem własnych klas
- Funkcje z parametrami domniemanymi
- Enkapsulacja
- Kwalifikatory dostępu do klas
- Metody specjalne: konstruktor, destruktor

# Pisanie własnych klas

# Własna klasa – atrybuty

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

class Tpralka
{
public:
    int nr_programu;
    double temp_prania;
    string  nazwa;
};


int main()
{
   Tpralka czerwona;

   return 0;
}
```

Do słowa **public** jeszcze dojdziemy!

Atrybuty klasy

No i mamy obiekt typu `Tpralka`

# Własna klasa – atrybuty cd.

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

class Tpralka
{
public:
    int nr_programu;
    double temp_prania;
    string  nazwa;
};

int main()
{
   Tpralka czerwona;
   Tpralka niebieska;
   Tpralka zielona;

   return 0;
}
```

Do słowa **public** jeszcze dojdziemy!

Atrybuty klasy

Więcej obiektów, ale jeszcze nie mają one określonego stanu.

# Wskaźnik i adres pamięci – przypomnienie!

**Wskaźnik** (ang. *pointer*) to specjalny rodzaj zmiennej, w której zapisany jest adres w pamięci komputera.

Oznacza to, że wskaźnik **wskazuje** miejsce, gdzie zapisana jest jakaś informacja (np. zmienna typu liczbowego, obiekt czy struktura).

**Adres pamięci** to liczba całkowita, jednoznacznie definiująca położenie pewnego obiektu w pamięci komputera. Tymi obiektami mogą być np. zmienne, elementy tablic czy nawet funkcje.

# Deklaracja wskaźników – przypomnienie!

```
TYP * zmienna;
```

```
int* p;
```

**'p'** jest zmienną wskaźnikową przechowującą adres liczby typu **'int'**

```
double* q;
```

**'q'** jest zmienną wskaźnikową przechowującą adres liczby typu **'double'**

# Ustawiamy wskaźnik – przypomnienie!

By stworzyć wskaźnik do zmiennej i móc się nim posługiwać, **należy przypisać** mu odpowiednią wartość - **adres obiektu**, na jaki chcieliśmy aby wskazywał

W języku C++ możemy "zapytać się" o adres za pomocą operatora **&** (operatora pobrania adresu).

```
int* w;
int a = 3;
int b = 5;

w = &a;

w = &b;
```

**'w'** jest wskaźnikiem na zmienną typu **int**

wskaźnik **'w'** wskazuje teraz na obszar pamięci zajmowany przez zmienną **'a'**

a teraz **'w'** wskazuje na zmienną **'b'**

# Referencje – przypomnienie!

**Referencja** działa trochę jak wskaźnik, który może wskazywać tylko na pojedynczą, określoną zmienną.

Tzn., do referencji można przypisać adres obiektu **tylko raz**, a dalsze używanie zmiennej referencyjnej niczym się nie różni od używania zwykłej zmiennej.

Operacje, jakie wykonamy na zmiennej referencyjnej, zostaną odzwierciedlone na zmiennej zwykłej, z której pobrano adres.

# Deklaracja referencji – przypomnienie!

```
TYP & referencja;
```

```
TYP innaZmienna;
```

Tworzymy zwykłą zmienną

```
TYP &referencja = innaZmienna;
```

Tworzymy **referencję do naszej zmiennej**. Musimy ją **<u>od razu zainicjalizować</u>**, tzn. pokazać do jakiej zmiennej tworzymy referencję.

# Użycie referencji – przypomnienie!

```
int main()
{
    int i = 0;
    int &ref_i = i;

    cout << i;       // wypisuje 0

    ref_i = 1;
    cout << i;       // wypisuje 1

    cout << ref_i;   // wypisuje 1

    return 0;
}
```

Zauważmy, że zmieniając wartość referencji, zmieniamy także wartość oryginalnej zmiennej!

Referencja jest „osobistym” wskaźnikiem na daną zmienną

*„Referencja jest przezwiskiem na zmienną!”**

**Jerzy Grębosz – „Pasja C++”*

# Odnoszenie się do składników obiektu

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

class Tpralka
{
public:
    int nr_programu;
    double temp_prania;
    string  nazwa;
};

int main()
{
   Tpralka czerwona;
   Tpralka* wskaz;
   Tpralka &ruda = czerwona;

   czerwona.temp_prania = 60;     // nazwą obiektu

   wskaz = &czerwona;
   wskaz -> temp_prania = 60;     // wskaźnikiem


   ruda.temp_prania = 60;         // referencją
   return 0;
}
```

Aby odnieść się do składników obiektu, możemy posługiwać się jedną z poniższych notacji:

- obiekt **.** *składnik*
- wskaźnik **->** *składnik*
- referencja **.** *składnik*

Definicja egzemplarza obiektu

Definicja wskaźnika do pralek

Definicja referencji (przezwiska pralki)

Jak odnieść się do składnika **temp_prania**?

# Zadanie 17.1

**Zad 17.1** Przepisz program wykorzystujący klasę **`Tpralka`**. Skompiluj kod. Następnie wykonaj polecenia:

- dodaj 2 pola z atrybutami klasy **`Tpralka`**
  - ładowność pralki w litrach – jako liczbę typu **`double`**
  - wysokość pralki w cm – jako liczbę typu **`int`**
- stwórz 3 obiekty klasy **`Tpralka`**
- w pierwszym stworzonym obiekcie ustaw `nr_programu` na 1 posługując się nazwą obiektu
- w drugim stworzonym obiekcie ustaw wartość ładowności pralki na 5.25 posługując się wskaźnikiem
- w trzecim stworzonym obiekcie ustaw nazwę pralki na „Frania” posługując się referencją
- w przypadku wcześniejszego zakończenia zadania spróbuj poustawiać inne parametry pralek wg własnego uznania przy pomocy dowolnie wybranej metody

# Metody klasy (funkcje składowe)

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

class Tpralka
{
public:

    void pierz (int program);
    void wiruj (int minuty);

    int nr_programu;
    double temp_prania;
    string  nazwa;

    int krochmalenie ();
};
```

Nieważne, w którym miejscu klasy zadeklarujemy zmienną lub funkcję – jest ona widoczna od razu w całej klasie!

Mówimy, że nazwy zadeklarowane w klasie mają **zakres ważności** równy obszarowi klasy.

Funkcje składowe (metody)

Dane składowe (atrybuty)

Znowu jakaś funkcja składowa

przyklad1.cpp

# Implementacja klasy w jednym pliku

# Zadanie 17.2

**Zad 17.2** Utwórz nowy projekt. Napisz kod, który będzie reprezentował Twoją wybraną klasę, którą wymyśliłeś na poprzednich zajęciach. W ramach tego zadania wykonaj następujące „kropki”:

- niech Twoja klasa ma chociaż 3 atrybuty, najlepiej niech będą różnych typów
- niech Twoja klasa ma 2 dowolne metody. Mogą one być typu **void**, jeśli nie mają zwracać żadnej wartości. Nie muszą mieć żadnych argumentów (chyba, że autor twierdzi inaczej). Jako implementację metody dla uproszczenia wyświetl na ekranie konsoli co ta metoda powinna robić. Np. jeśli masz klasę **Samochod** a w nim metodę **jedz()** to przykładowa implementacja może wyglądać następująco:

```
void jedz()
{
    cout << ”Jade!” << endl;
}
```

- stwórz 2 obiekty Twojej klasy
- spróbuj nadać wartości poszczególnym atrybutom obiekt posługując się różnym poznanym metodom (bezpośrednio przez nazwę obiektu, wskaźnik lub referencję)

Implementację klasy wykonaj w jednym pliku źródłowym `main.cpp`.

# Funkcje z parametrami domniemanymi

Prototypy (jak również definicje) funkcji mogą wykorzystywać parametry formalne o wartościach domniemanych.

Postać prototypu:

```
typ funkcja (typ po1, typ po2, … , typ pd1=wd1, typ pd2=wd2);
```

gdzie:

- **po1, po2** – parametry obowiązkowe (ale może ich też nie być wcale)
- **pd1, pd2** – parametry domniemane o predefiniowanych wartościach: **wd1**, **wd2**

Parametry domniemane mogą występować tylko po parametrach obowiązkowych:

```
long suma (int a, int b=10);      //dobrze

long suma (int a=10, int b);      //źle
```

# Funkcje z parametrami domniemanymi cd.

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

long potega (int podstawa=2, int wykładnik=2)
{
    long wynik=podstawa;

    for (int i=1; i<wykładnik; i++) wynik *= podstawa;

    return wynik;
}

int main ()
{
    cout << potega() << endl;
    cout << potega(3) << endl;
    cout << potega(3, 4) << endl;
}
```

Jaki będzie wynik?

4
9
81

# Zadanie 17.3

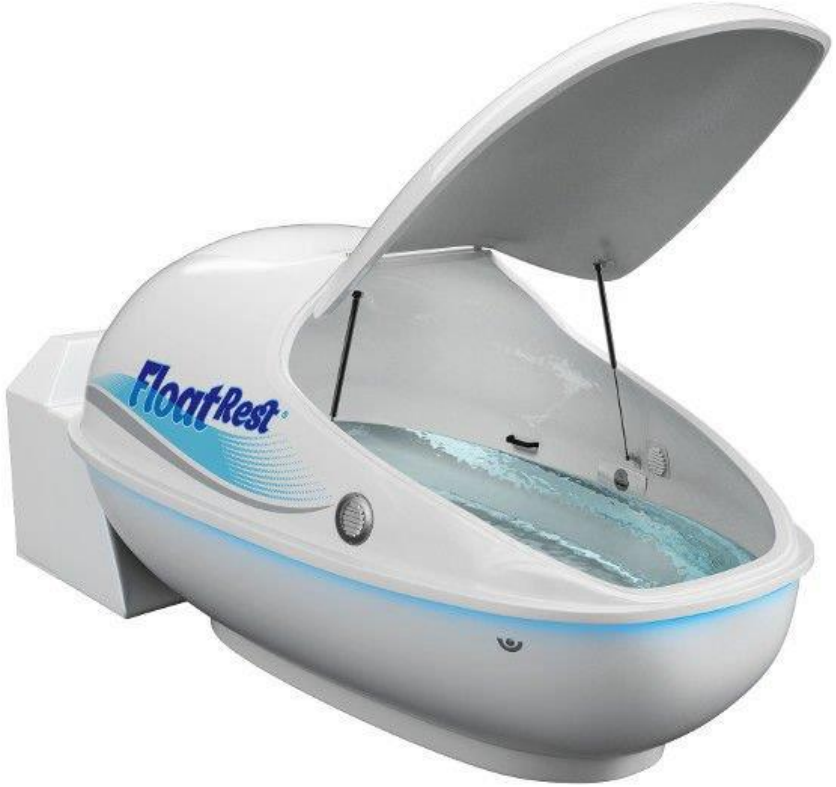
**Zad 17.3** Utwórz nowy projekt. Napisz kod, który będzie reprezentował klasę `Kalkulator`:

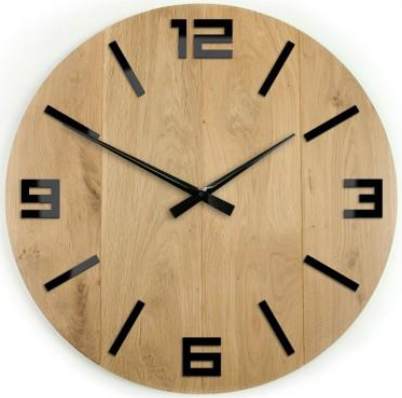
- klasa ta nie będzie posiadała żadnych atrybutów, będzie miała tylko publiczne metody
- zaimplementuj metody:
  - dodawanie
  - odejmowanie
  - mnożenie
  - dzielenie
- metody, które zaimplementujesz muszą mieć zdefiniowane parametry domyślne (ustawione na wartość 1 lub 0, zgodnie z matematycznym sensem)

# Założenia paradygmatu OOP

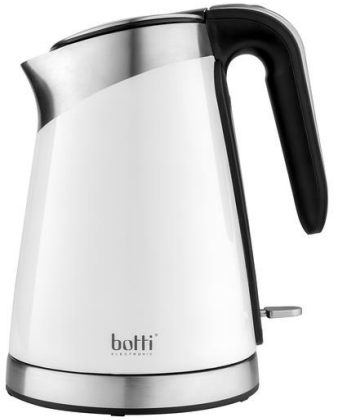
- Abstrakcja
- Hermetyzacja (kapsułkowanie)
- Dziedziczenie
- Polimorfizm

# Enkapsulacja (kapsułkowanie)




nastaw()




- Zakresem ważności jednej funkcji nastaw jest klasa czajnik.
- Zakresem ważności innej funkcji nastaw jest zegarek.

# Enkapsulacja cd.

- **Ukrywanie** pewnych szczegółów implementacyjnych, danych i metod obiektu

- Ukrywanie przed kim, przed czym?
  - Czasem przed innymi programistami, którzy będą korzystać z przygotowanych przez nas klas i obiektów, a czasem przed pozostałymi elementami tej samej aplikacji

- Ułatwia podział pracy i pracę zespołową
  - Graficy przygotowujący szatę graficzną dla sklepu internetowego wcale nie muszą wiedzieć jakich algorytmów używamy np. do obliczania rabatu, interesuje ich tylko jak można odczytać wartość rabatu dla konkretnego produktu i jak ją wyświetlić na stronie WWW (interfejs).

# Enkapsulacja cd.

- Jesteśmy przyzwyczajeni do tego, że do obsługi czasem bardzo skomplikowanych urządzeń (jak np. telewizor, radio) wystarczy krótka instrukcja i np. pilot.

- Nie potrzebna jest nam wiedza na temat konstrukcji telewizora, ważne aby się włączał, wyłączał, zmieniał kanały, zmieniał poziom głośności itd.

- **Uproszczenie** obsługi, ograniczenie liczby dostępnych funkcji do niezbędnego minimum – te czynniki zmniejszają ryzyko nieumyślnego zepsucia urządzenia – to samo dotyczy tworzonego przez nas oprogramowania!

# Kwalifikatory dostępu do klas

Lista elementów składowych w definicji klasy może zostać podzielona na sekcje określające poziom dostępu przy użyciu słów kluczowych: **private**, **public** i **protected**.

- etykieta **private**: deklarowane za nią składniki (funkcje i dane) są dostępne **tylko z wnętrza klasy** (także dla funkcji zaprzyjaźnionych z klasą);
- etykieta **public**: deklarowane za nią składniki (funkcje i dane) są dostępne **bez ograniczeń**;
- etykieta **protected**: deklarowane za nią składniki (funkcje i dane) są dostępne **z wnętrza klasy oraz dla klas potomnych.**

# Kwalifikatory dostępu do klas cd.

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

class Tnasz_typ
{
private:
     int liczba;
     double temperatura;
     string  komunikat;

     bool czy_gotowe();

public:
     double prekosc;
     int zrob_pomiar();
};
```

prywatne dane składowe

prywatna funkcja składowa

publiczna dana składowa

publiczna funkcja składowa

# Domniemany dostęp do składników

Zakłada się, że dopóki w definicji klasy nie wystąpi żadna z etykiet, składniki przez **domniemanie** mają określony dostęp (**private** lub **public**).

UWAGA:

- w klasie zdefiniowanej przy użyciu słowa kluczowego **struct** domyślnie wszystkie elementy składowe są niechronione (mają status - **public**);
- w klasie zdefiniowanej przy użyciu słowa kluczowego **class** domyślnie wszystkie elementy składowe są chronione (mają status - **private**).

# struct vs class

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

class Student
{
   string  imie;
};

int main ()
{
   Student stud1;
   stud1.imie = "Jan";
}
```

Tu będzie błąd podczas kompilacji.

*'imie' is a private member of 'Student'*

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

struct Student
{
   string  imie;
};

int main ()
{
   Student stud1;
   stud1.imie = "Jan";
}
```

Więcej na temat domniemanych etykiet:

*Przyklad2*

# Kilka porad podczas pisania klas

Porady przy pracy nad większymi projektami:

- Na górze definicji klasy umieszcza się najpierw składnik **public**, bo te są najbardziej interesujące dla późniejszego użytkowania klasy;
- Niżej są składniki **protected** (porozmawiamy o nich w stosownej chwili);
- Najniżej są składniki **private**, bo te już interesują tylko tego, kto pracuje nad klasą (a nie jedynie ją użytkuje);
- Czasem składniki **private** umieszcza się na górze klasy, ponieważ domniemana etykieta dla **class** jest właśnie prywatna;
- W przypadku klas, które nie wchodzą w relacje dziedziczenia, kwalifikator **protected** nie powinien być używany (jest równoważny **private**).

# Inicjalizacja wartości w klasie

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

class Pionek
{
    int  pozycja {0};
    double  stan_zdrowia {1.0};
    int  numer_mojej_druzyny;

};
```

Inna forma

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

class Pionek
{
    int  pozycja = 0;
    double  stan_zdrowia = 1.0;
    int  numer_mojej_druzyny;

};
```

Tego rodzaju wstępne nadanie wartości składnikowi nazywa się inicjalizacją „w klasie” (ang. in-class).

Można ją kompilatorowi zlecić, stosując albo zapis za pomocą klamer { } , albo zapis ze znakiem =.

# Gettery i settery

- wartości pól pobieramy metodami typu: getter;
- wartości pól ustawiamy metodami typu: setter i/lub w konstruktorze;

`Przyklad3`

# Metody specjalne - konstruktor

**Konstruktor** to metoda wywoływana automatycznie w momencie tworzenia obiektu.

Metoda ta może być przeciążana np.:

- `Klasa();` - postać tzw. konstruktora domyślnego,
- `Klasa(typ wartosc);` - postać tzw. konstruktora jednoparametrowego,

- służy do inicjacji pól obiektów,
- nazwa konstruktora pokrywa się z nazwą klasy,
- w przypadku tej metody nie podaje się typu zwracanego wyniku,
- umieszcza się ją w części publicznej definicji klasy (choć zdarzają się wyjątki),

# Metody specjalne - destruktor

Metoda wywoływana automatycznie w momencie niszczenia obiektu.

- służy do wykonania niezbędnych czynności przed likwidacją obiektu,
- nazwa destruktora ma postać: nazwa klasy poprzedzony znakiem ~
- w przypadku tej metody nie podaje się typu zwracanego wyniku;
- umieszcza się ją w części publicznej definicji klasy, w klasie istnieje tylko jeden destruktor.

*Przyklad3*

# Zadanie 17.4

**Zad 17.4** Utwórz nowy projekt. Napisz program, który posłuży do sterowania samochodem i motocyklem:

- Klasa **`Samochod`** powinna mieć takie atrybuty jak
  - `mocSilnika` – liczba całkowita nieujemna. Domyślnie jest to 125KM.
  - `pojemnoscSilnika` – liczba całkowita nieujemna. Domyślnie jest to 1500 $cm^3$.
  - `silnik` – wartość `true` or `false` informująca czy auto ma uruchomiony silnik. Domyślnie jest to wartość false
  - `jazda` – wartość `true` or `false` informująca czy auto jedzie. Domyślnie jest to wartość `false`.
- Klasa **`Samochod`** powinna mieć takie metody jak:
  - Konstruktor inicjujący domyślnie wartości;
  - `void hamuj()` – metoda wyświetla informację na ekranie, że auto hamuje tylko wtedy, gdy wykryje uruchomiony silnik w samochodzie. W przeciwnym wypadku wyświetla informację, że samochód nie jedzie;
  - `void uruchomSilnik()` – wyświetla informację na ekranie, że uruchomiono silnik oraz uruchamia silnik (zmienia wartość pola silnik na `true`);
  - `void jedz()` – jeśli jest włączony silnik to wypisuje na ekranie informację o tym, że auto się porusza oraz umożliwia jazdę (zmienia wartość pola jazda na `true`);
  - `void wyświetl()` – wyświetla informacje o danych samochodu (jego moc oraz pojemność silnika).
- Klasa **`Motocykl`** powinna mieć takie atrybuty jak
  - `mocSilnika` – liczba całkowita nieujemna. Domyślnie jest to 20KM.
  - `pojemnoscSilnika` – liczba całkowita nieujemna. Domyślnie jest to 700 $cm^3$.
- Klasa **`Motocykl`** powinna mieć takie metody jak:
  - Konstruktor inicjujący domyślnie wartości;
  - `void wyświetl()` – wyświetla informacje o danych samochodu (jego moc oraz pojemność silnika).
- Po uruchomieniu programu wyświetl użytkownikowi menu główne, w którym będzie on mógł uruchomić silnik samochodu, jechać autem, wyświetlić informacje o samochodzie, wyświetlić informacje o motocyklu lub wyjść z programu.
- Utwórz po jednym egzemplarzu obiektu. Na podstawie wyboru użytkownika niech zostaną wywoływane odpowiednie metody klas **`Samochod`** oraz **`Motocykl`**. Możesz wykorzystać instrukcję wyboru `switch`.
- Całość instrukcji `switch` (lub serii `if else`) umieść w jednej pętli nieskończonej `while`, aby program działał tak długo, jak tego chce użytkownik.
- Spróbuj samemu rozwiązać problem wyjścia z nieskończonej pętli!