



Podstawy Programowania

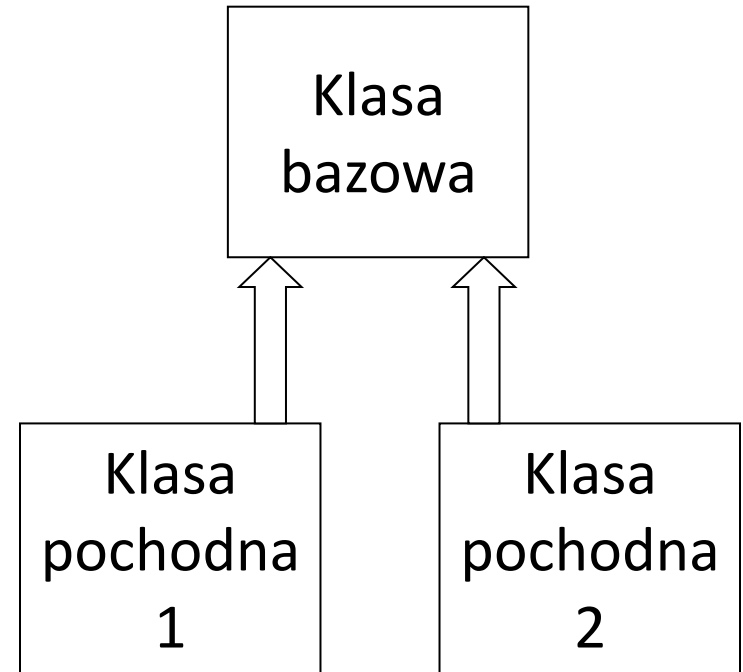
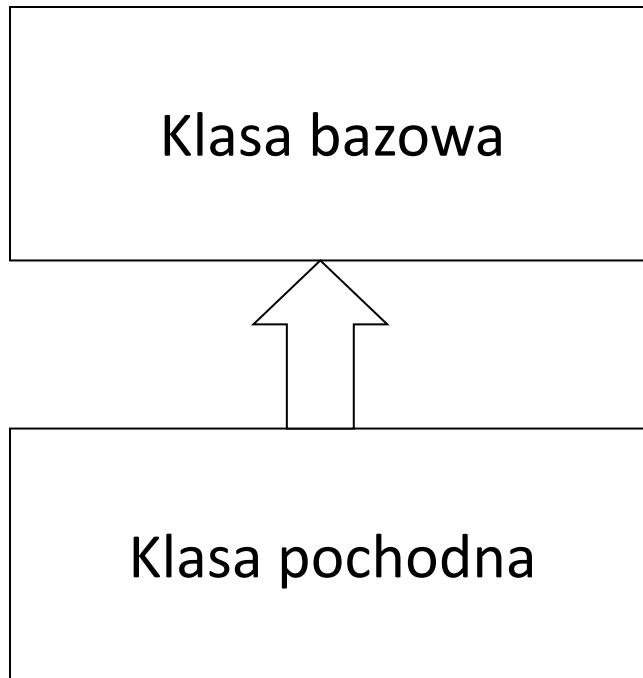
dr inż. Tomasz Marciniak

- 
- Dziedziczenie
 - klasy

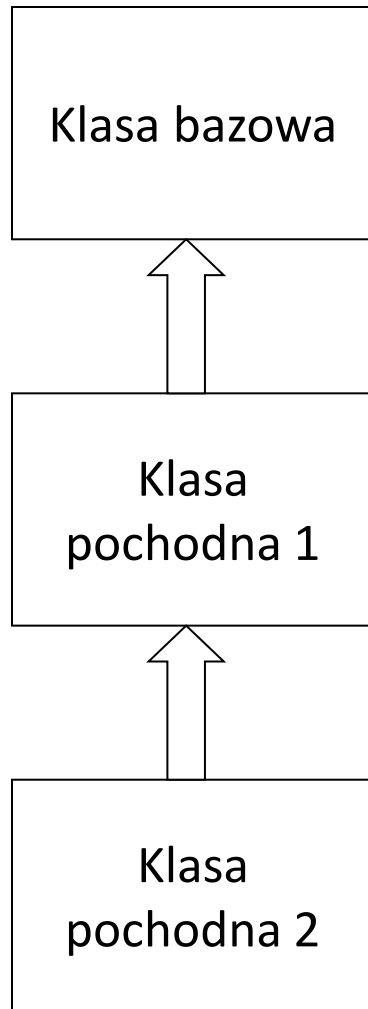
Dziedziczenie

- Umożliwia wykorzystanie istniejącej klasy (lub klas) jako bazę dla nowej klasy;
- W C++ można korzystać z tzw. Wielokrotnego dziedziczenia, czyli tworzenia nowej klasy w oparciu o kilka klas bazowych;
- Jest kluczowym mechanizmem dla klas.

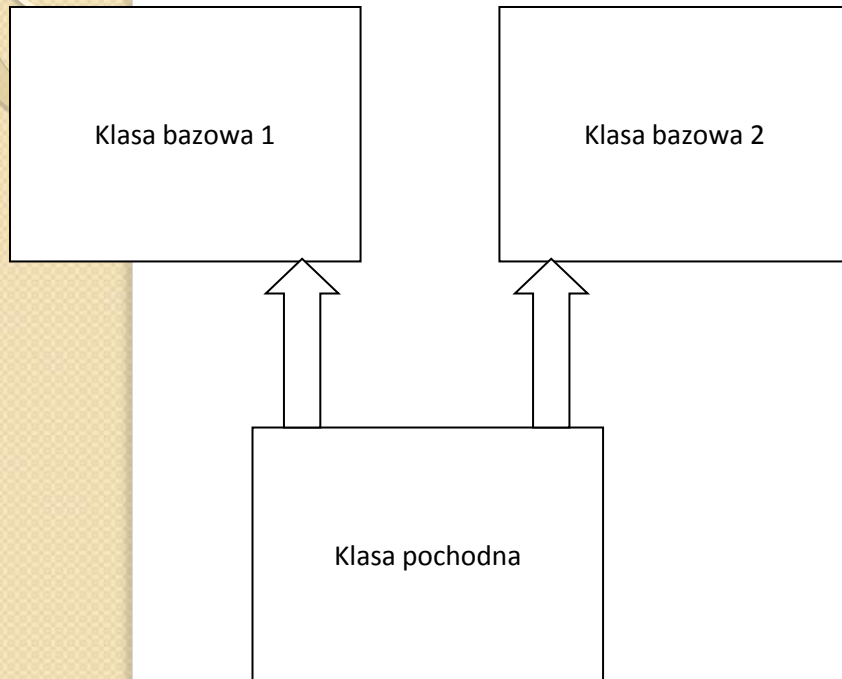
Dziedziczenie proste



Dziedziczenie proste kaskadowe



Dziedziczenie mnogie



Notacja UML (Unified Modeling Language)

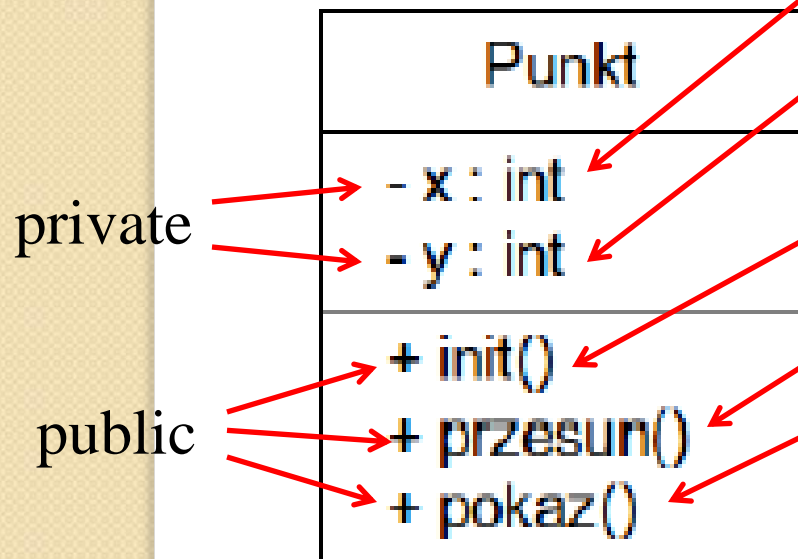
Nazwa klasy
Dane
Metody

Punkt
- x : int - y : int
+ init() + przesun() + pokaz()

```
class Punkt
{
private :
int x ;
int y ;
public :
void init ( int , int );
void przesun (int, int );
void pokaz ( ) ;
} ;
```

Notacja UML (Unified Modeling Language)

Nazwa klasy
Dane
Metody



class Punkt

{

private :

int x ;

int y ;

public :

void init (int ,int) ;

void przesun(int ,int);

void pokaz () ;

} ;

Przykład – klasa punkt

```
1  #include <iostream>
2  #include <math.h>
3  #include <conio.h>
4  using namespace std;
5  class punkt{
6      double x , y ;
7      public :
8      void init(double xx=0.0,double yy=0.0)
9      {
10         x = xx;
11         y = yy ;
12     }
13     void pokaz ()
14     { cout << "wspolrzedne :  " << x << "  " << y << endl ;
15     }
16     double wsX() { return x ; }
17     double wsY() { return y ; }
18 }
```

Przykład – klasa punktB

Dziedziczymy z klasy punkt

```
20 class punktB : public punkt{  
21     public :  
22     double promien()  
23     { return sqrt(wsX() * wsX() + wsY() * wsY()); }  
24 } ;  
25  
26 int main (){  
27     punktB a ;  
28     a . init(3, 4) ;  
29     a . pokaz() ;  
30     cout << "promien          : " << a . promien() ;  
31     getch() ;  
32     return 0 ;  
33 }
```

Przykład – klasa punktB

Dodajemy metodę
obliczającą promień

```
20 class punktB : public punkt{
21     public :
22     double promien()
23         { return sqrt(wsX() * wsX() + wsY() * wsY()); }
24 } ;
25
26 int main (){
27     punktB a ;
28     a . init(3, 4) ;
29     a . pokaz() ;
30     cout << "promien" : " << a . promien() ;
31     getch() ;
32     return 0 ;
33 }
```

Przykład – klasa punktB

```
20 class punktB : public punkt{  
21     public :  
22     double promien()  
23     { return sqrt(wsX() * wsX() + wsY() * wsY()); }  
24 } ;  
25  
26 int main (){  
27     punktB a ;  
28     a . init(3, 4) ;  
29     a . pokaz() ;  
30     cout << "promien          :  " << a . promien() ;  
31     getch() ;  
32     return 0 ;  
33 }
```

Definiujemy obiekt a

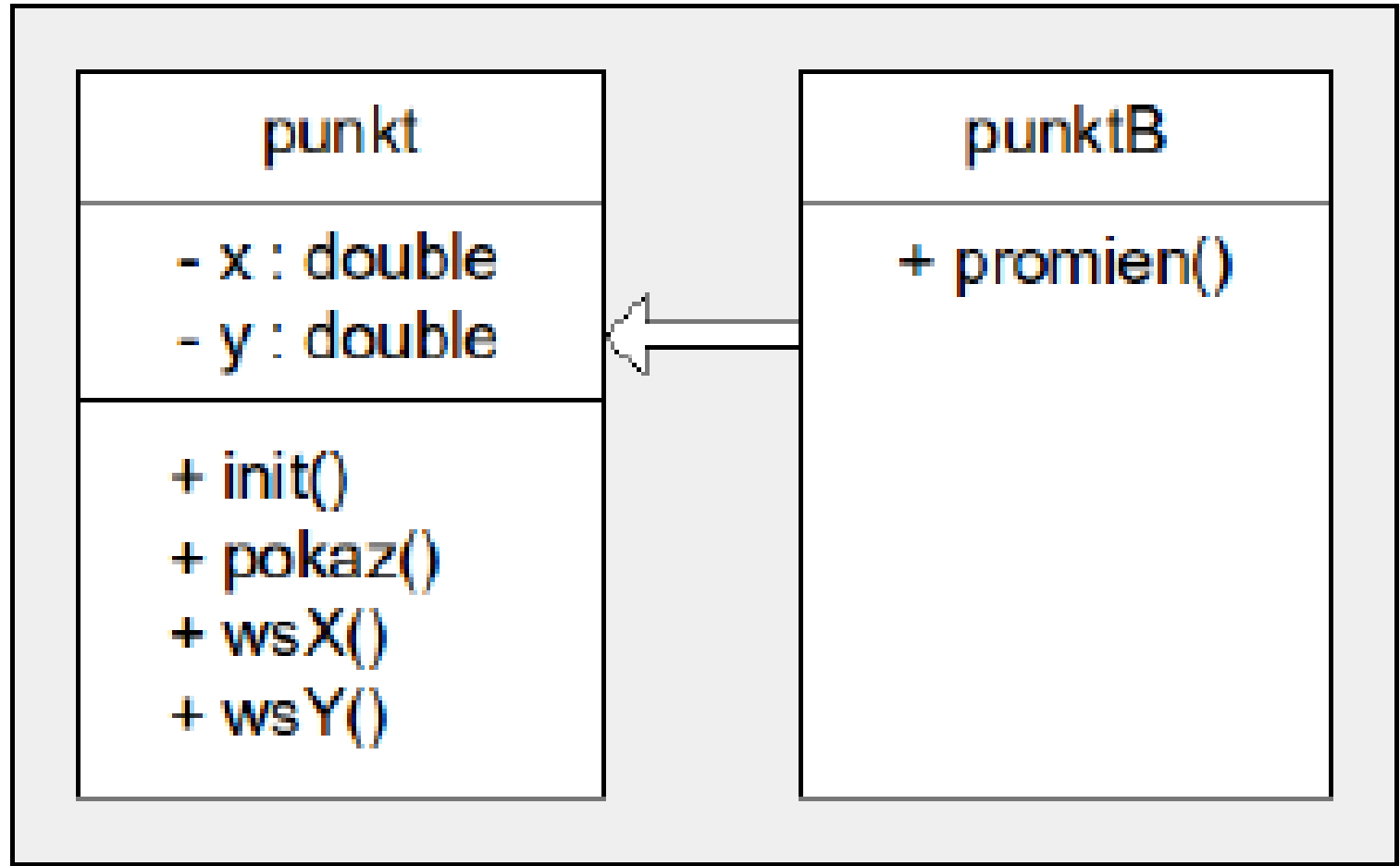
Przykład – klasa punktB

```
20 class punktB : public punkt{
21     public :
22     double promien()
23         { return sqrt(wsX() * wsX() + wsY() * wsY()); }
24 } ;
25
26 int main (){
27     punktB a ;
28     a . init(3, 4) ;
29     a . pokaz() ;
30     cout << "promien" : " << a . promien() ;
31     getch() ;
32     return 0 ;
33 }
```

Korzystamy z metody
klasy bazowej init i
pokaz

```
wspolrzedne : 3 4
promien      : 5
```

Diagram klas w UML

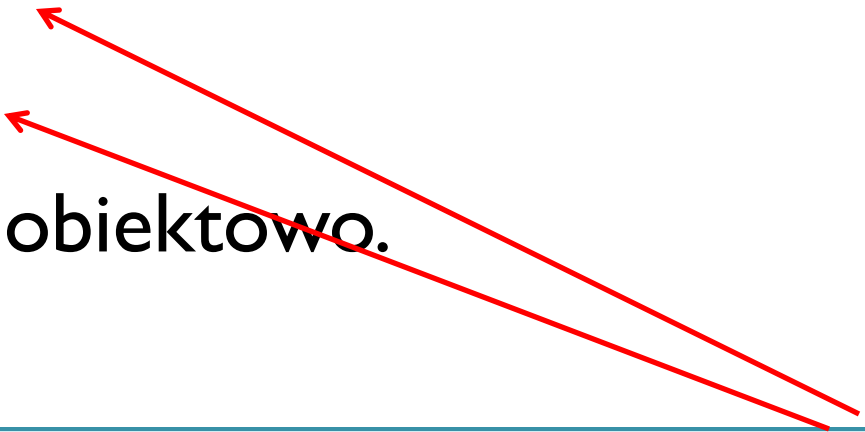


Metody programowania w C++

- Proceduralne,
- Strukturalne,
- Zorientowane obiektowo.

Metody programowania w C++


- Proceduralne,
- Strukturalne,
- Zorientowane obiektowo.



**•Program traktowany jako seria instrukcji i procedur działających na danych,
•Dane całkowicie odseparowane od procedur.**

Metody programowania w C++

- Proceduralne,
- Strukturalne,
- Zorientowane obiektowo.



**•Stosowane do dużych programów,
•Dostarcza technik zarządzania
złożonymi elementami,
•Łączy w logiczną całość dane i
funkcje.**

Programowanie obiektowe

- **W podejściu obiektowym**
 - koncentrujemy się na obiekcie postrzeganym przez użytkownika, na danych potrzebnych do opisanie takiego obiektu i operacjach opisujących interakcje użytkownika z obiektem.
- **Czym jest abstrakcja?**
 - Abstrakcja polega na wyodrębnieniu i uogólnieniu najważniejszych cech problemu i wyrażeniu rozwiązania właśnie w obrębie tych cech.
- **Czym jest interfejs?**
 - Zaimplementowaną możliwością komunikacji użytkownika z obiektem, strukturą.

Struktura

```
struct produkt {  
    int waga;  
    float cena;  
};  
produkt jablko;  
produkt banan, melon;
```

inny sposób:

```
struct produkt {  
    int waga;  
    float cena;  
} jablko, banan, melon;
```

- Struktury pozwalają na przechowywanie danych różnego typu, zebranych po jedną nazwą.
- Dostęp do pól wewnątrz struktury:
 - jablko.waga=1;
 - banan.cena=4.35;

Obiekty

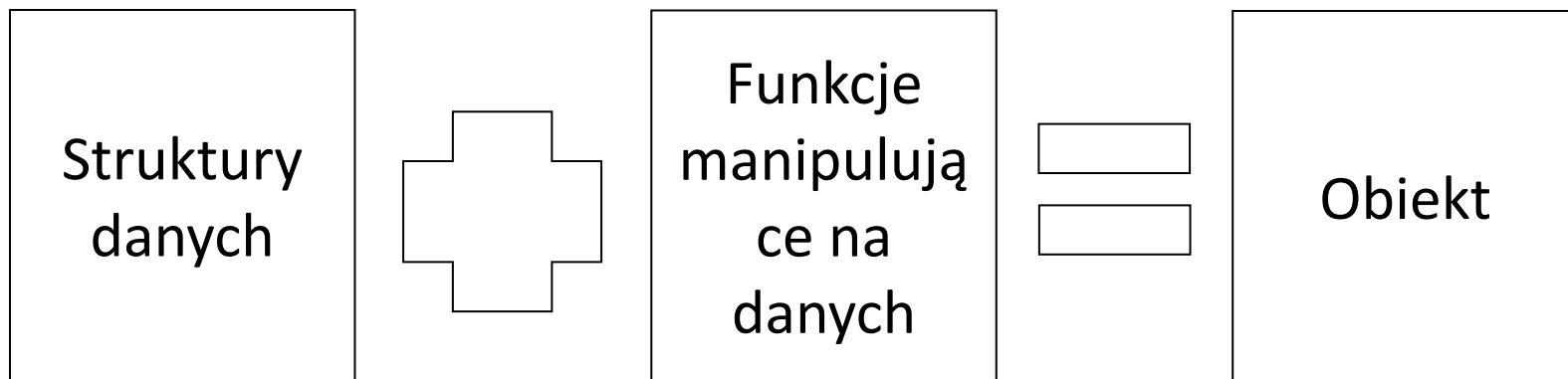
Struktura

- pozwala na przechowywanie różnych typów pól;
- **nowy typ danych**;

Klasa

- pozwala na przechowywanie różnych typów pól;
- pozwala na definiowanie zachowań obiektu – funkcji składowych;
- Nowy **typ** pozwalający na definiowanie obiektów.

Obiekty



Obiekty i klasy

Obiekt składa się z:

- Danych (atrybuty/elementy),
- Operacje (działania/metody/funkcje).

Klasa:

- Opisuje obiekt,
- Formalnie to typ,
- Zmienne typu określonego przez klasę to instancje / obiekty.

Deklaracja klasy

```
class nazwa_typu{  
    //ciało klasy  
    // zmienne i funkcje  
};
```

Dostęp do danych
składowych

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

```
class pralka{  
private:  
    int  nr_programu;  
    float temp_prania;  
    char nazwa[80];  
};  
  
pralka czerwona; //obiekt  
pralka * wsk; //wskaźnik  
pralka & ruda = czerwona;  
//referencja
```

Deklaracja klasy

Nazwa klasy

Słowo kluczowe

```
class nazwa_typu{
```

```
//ciało klasy
```

```
};
```

Miejsce na dane
i metody

```
class pralka{
```

```
private:
```

```
int nr_programu;
```

```
float temp_prania;
```

```
char nazwa[80];
```

```
};
```

```
pralka czerwona; //obiekt
```

```
pralka * wsk; //wskaźnik
```

```
pralka & ruda = czerwona;  
//referencja
```

Dostęp do danych
składowych

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

Deklaracja klasy

Nazwa klasy

Słowo kluczowe

class nazwa_typu{

//ciało klasy

};

Miejsce na dane
i metody

Dostęp do danych
składowych

o obiekt składnik

**Deklaracja klasy
nie rezerwuje
miejsca w pamięci
na nią**

class pralka{

private:

int nr_programu;

float temp_prania;

char nazwa[80];

};

pralka czerwona; *//obiekt*

pralka * wsk; *//wskaźnik*

pralka & ruda = czerwona;
//referencja

Po co więc deklaracja klasy?

Informuje kompilator o:

- Czym jest typ opisany przez klasę,
- Jakie dane zawiera (zmienne składowe),
- Co może robić (metody),
- Ile miejsca w pamięci trzeba przygotować w przypadku tworzenia zmiennej.

Deklaracja klasy

```
class nazwa_typu{  
    //ciało klasy  
    // zmienne i funkcje  
};
```

```
class pralka{  
private:  
    int  nr_programu;  
    float temp_prania;  
    char nazwa[80];  
};
```

Definicja obiektu
(tak jak każda inna zmienna)

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

pralka czerwona; //obiekt
*pralka * wsk; //wskaźnik*
pralka & ruda = czerwona;
//referencja

Klasy i składowe

- Nowy typ zmiennych tworzy się, deklarując **klasę**.
- Klasa jest grupą zmiennych — często o różnych typach — skojarzonych z zestawem odnoszących się do nich funkcji.
- Klasa może składać się z dowolnej kombinacji zmiennych prostych oraz zmiennych innych klas. Zmienna wewnątrz klasy jest nazywana **zmienną składową** lub **daną składową**.

Przykład klasy

- Klasa `Car` (samochód) może posiadać składowe reprezentujące: siedzenia, typ radia, opony, itd.
- Zmienne składowe są zmiennymi w danej klasie. Stanowią one część klasy, tak jak *koła* i *silnik* stanowią część samochodu.

Przykład

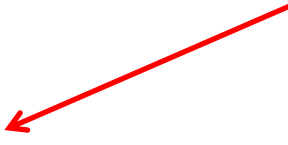
```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      public:
6          int a,b;
7  };
8
9  int main(){
10     kwadrat kw1;
11     kwadrat kw[5];
12     kwadrat *wskkw;
13     kw1.a=1; kw1.b=2; kw[0].a=3; kw[0].b=4; wskkw=&kw1;
14     cout << "dla kw1 a=" << kw1.a << " b=" << kw1.b;
15     cout << "\ndla kw[0] a=" << kw[0].a << " b=" << kw[0].b;
16     cout << "\ndla wskkw a=" << wskkw->a << " b=" << wskkw->b;
17     getch();
18     return 0;
19 }
```

Deklaracja
klasy „kwadrat”

Przykład

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      public:
6          int a,b;
7  };
8
9  int main(){
10     kwadrat kw1;
11     kwadrat kw[5];
12     kwadrat *wskkw;
13     kw1.a=1; kw1.b=2; kw[0].a=3; kw[0].b=4; wskkw=&kw1;
14     cout << "dla kw1 a=" << kw1.a << " b=" << kw1.b;
15     cout << "\ndla kw[0] a=" << kw[0].a << " b=" << kw[0].b;
16     cout << "\ndla wskkw a=" << wskkw->a << " b=" << wskkw->b;
17     getch();
18     return 0;
19 }
```

Definicja obiektu kw1



Przykład

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      public:
6          int a,b;
7  };
8
9  int main(){
10     kwadrat kw1;
11     kwadrat kw[5];
12     kwadrat *wskkw;
13     kw1.a=1; kw1.b=2; kw[0].a=3; kw[0].b=4; wskkw=&kw1;
14     cout << "dla kw1 a=" << kw1.a << " b=" << kw1.b;
15     cout << "\ndla kw[0] a=" << kw[0].a << " b=" << kw[0].b;
16     cout << "\ndla wskkw a=" << wskkw->a << " b=" << wskkw->b;
17     getch();
18     return 0;
19 }
```

Definicja tablicy
obiektów „kwadrat”



Przykład

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      public:
6          int a,b;
7  };
8
9  int main(){
10     kwadrat kw1;
11     kwadrat kw[5];
12     kwadrat *wskkw;
13     kw1.a=1; kw1.b=2; kw[0].a=3; kw[0].b=4; wskkw=&kw1;
14     cout << "dla kw1 a=" << kw1.a << " b=" << kw1.b;
15     cout << "\ndla kw[0] a=" << kw[0].a << " b=" << kw[0].b;
16     cout << "\ndla wskkw a=" << wskkw->a << " b=" << wskkw->b;
17     getch();
18     return 0;
19 }
```

Definicja wskaźnika na
obiekt typu „kwadrat”

Przykład

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      public:
6          int a,b;
7  };
8
9  int main(){
10     kwadrat kw1;
11     kwadrat kw[5];
12     kwadrat *wskkw;
13     kw1.a=1; kw1.b=2; kw[0].a=3; kw[0].b=4; wskkw=&kw1;
14     cout << "dla kw1 a=" << kw1.a << " b=" << kw1.b;
15     cout << "\ndla kw[0] a=" << kw[0].a << " b=" << kw[0].b;
16     cout << "\ndla wskkw a=" << wskkw->a << " b=" << wskkw->b;
17     getch();
18     return 0;
19 }
```

Odwołanie do obiektu
statycznego

Przykład

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      public:
6          int a,b;
7  };
8
9  int main(){
10     kwadrat kw1;
11     kwadrat kw[5];
12     kwadrat *wskkw;
13     kw1.a=1; kw1.b=2; kw[0].a=3; kw[0].b=4; wskkw=&kw1;
14     cout << "dla kw1 a=" << kw1.a << " b=" << kw1.b;
15     cout << "\ndla kw[0] a=" << kw[0].a << " b=" << kw[0].b;
16     cout << "\ndla wskkw a=" << wskkw->a << " b=" << wskkw->b;
17     getch();
18     return 0;
19 }
```

Odwołanie do tablicy
obiektów

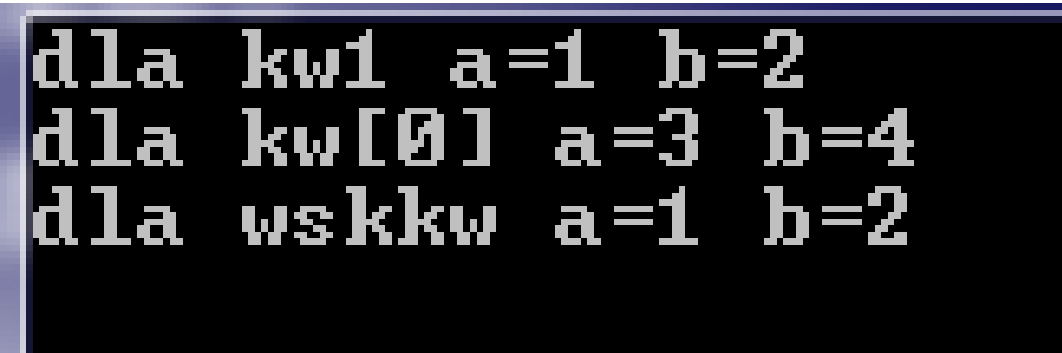
Przykład

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      public:
6          int a,b;
7  };
8
9  int main(){
10     kwadrat kw1;
11     kwadrat kw[5];
12     kwadrat *wskkw;
13     kw1.a=1; kw1.b=2; kw[0].a=3; kw[0].b=4; wskkw=&kw1;
14     cout << "dla kw1 a=" << kw1.a << " b=" << kw1.b;
15     cout << "\ndla kw[0] a=" << kw[0].a << " b=" << kw[0].b;
16     cout << "\ndla wskkw a=" << wskkw->a << " b=" << wskkw->b;
17     getch();
18     return 0;
19 }
```

Odwołanie do obiektu dynamicznego

Przykład

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      public:
6          int a,b;
7  };
8
9  int main(){
10     kwadrat kw1;
11     kwadrat kw[5];
12     kwadrat *wskkw;
13     kw1.a=1; kw1.b=2; kw[0].a=3; kw[0].b=4; wskkw=&kw1;
14     cout << "dla kw1 a=" << kw1.a << " b=" << kw1.b;
15     cout << "\ndla kw[0] a=" << kw[0].a << " b=" << kw[0].b;
16     cout << "\ndla wskkw a=" << wskkw->a << " b=" << wskkw->b;
17     getch();
18     return 0;
19 }
```



```
dla kw1 a=1 b=2
dla kw[0] a=3 b=4
dla wskkw a=1 b=2
```

Interfejs publiczny klasy

- **Interfejs publiczny** (szkielet interakcji między dwoma systemami) składa się z metod klasy implementowanych przez twórcę klasy.
- **Metody** są niezbędne do tego by wykonywać akcje, zgodne z zadaniami programu, zmieniać dane prywatne czy pozwalać na dostęp do zmiennych i metod
- **Metody** są częścią interfejsu publicznego pomiędzy obiektem klasy a użytkownikiem tego obiektu – np. innymi funkcjami programu.

Funkcje składowe

```
class pralka{  
public:  
    void pierz(int  
program);  
    void wiruj(int minuty);  
    int nr_programu;  
    float temp_prania;  
    char nazwa[80];  
    int krochmalenie(void):
```

Funkcje w danej klasie zwykle manipulują zmiennymi składowymi. Funkcje klasy nazywa się *funkcjami składowymi* lub *metodami klasy*.

- W definicji klasy – funkcje są wymieszane z danymi;
- Niezależnie od miejsca zdefiniowania składnika wewnątrz klasy – jest on znany w całej definicji klasy;
- Nazwy deklarowane w klasie mają zakres ważności równy obszarowi całej klasy.

Dane prywatne i publiczne

- W deklaracji klasy używanych jest także kilka innych słów kluczowych. Dwa najważniejsze z nich to: `public` (publiczny) i `private` (prywatny).

Kwantyfikatory dostępu

Określają dostęp do poszczególnych składników klasy z zewnątrz (czyli z programu głównego):

- **public**
 - Składnik public jest dostępny bez ograniczeń. Zwykle składnikami takimi są wybrane funkcje składowe (metody), które służą do modyfikowania określonych wartości private.
- **private**
 - Składnik private jest dostępny tylko dla funkcji składowych określonej klasy. Służy on do ukrywania zmiennych i funkcji wewnątrz obiektu.
- **protected**
 - Składnik protected jest dostępny tak jak private, z tą różnicą, że jest on dostępny również dla klas potomnych (dziedziczenie cech)

Przykład – do zastanowienia

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      private:
6          int a,b;
7  };
8
9  int main(){
10     kwadrat kw1;
11     kw1.a=1; kw1.b=2;
12     return 0;
13 }
```

Przykład – do zastanowienia

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      private:
6          int a,b;
7  };
8
9  int main(){
10     kwadrat kw1;
11     kw1.a=1; kw1.b=2;
12     return 0;
13 }
```

Błąd kompilatora,
zmiennne prywatne.

Co możemy zrobić?

Przykład – do zastanowienia

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      public:
6          int a,b;
7  };
8
9  int main(){
10     kwadrat kw1;
11     kw1.a=1; kw1.b=2;
12     return 0;
13 }
```

Zmieniamy na public
LUB

Przykład – do zastanowienia

```
1  #include <iostream>
2  #include <conio.h>
3  using namespace std;
4  class kwadrat{
5      private:
6          int a,b;
7      public:
8          SetAB(int x,int y){a=x;b=y;}
9  };
10
11  int main(){
12      kwadrat kw1;
13      kw1.SetAB(1,2);
14      return 0;
15  }
```

Dodajemy metodę

Oznaczanie danych składowych jako prywatnych – dobre praktyki

- Powinieneś przyjąć jako ogólną regułę, że dane składowe klasy należy utrzymywać jako prywatne.
- W związku z tym musisz stworzyć publiczne funkcje składowe, zwane funkcjami dostępowymi lub *akcesorami*. Funkcje te umożliwią odczyt zmiennych składowych i przypisywanie im wartości.
- Te funkcje **dostępowe** (akcesory) są funkcjami składowymi, używanymi przez inne części programu w celu odczytywania i ustawiania prywatnych zmiennych składowych.

Akcesory

- Publiczny akcesor jest funkcją składową klasy, używaną albo do odczytu wartości prywatnej zmiennej składowej klasy, albo do ustawiania wartości tej zmiennej.
- Akcesory umożliwiają oddzielenie szczegółów przechowywania danych klasy od szczegółów jej używania. Dzięki temu możesz **zmieniać sposób przechowywania danych klasy** bez konieczności przepisywania funkcji, które z tych danych korzystają.

Ważne

- Definicja klasy nie definiuje obiektów!!
- Definicja konkretnych egzemplarzy tej klasy sprawia, że powstają obiekty w pamięci !!
- Klasa to TYP a nie sam OBIEKT

Definiowanie funkcji składowych

- **Wewnątrz definicji klasy:**

```
class osoba {  
    public:  
        string imie;  
        string nazwisko;  
        int wiek;  
    osoba(string, string , int );  
    ~osoba(void);  
    void przedstawSie()  
    {  
        cout << "Nazywam sie " <<  
        imie << " " << nazwisko <<  
        endl;  
        cout << "mam " << wiek << "  
        lat." << endl;  
    };  
};
```

- **Na zewnątrz definicji klasy:**

```
class osoba {  
    public:  
        string imie;  
        string nazwisko;  
        int wiek;  
        void przedstawSie();//funkcja  
};  
void osoba::przedstawSie()  
{  
    cout << "Nazywam sie " << imie  
        << " " << nazwisko << endl;  
    cout << "mam " << wiek << "  
        lat." << endl;  
}
```

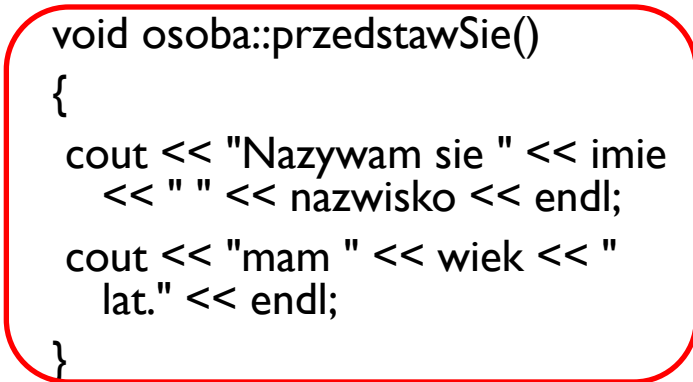
Definiowanie funkcji składowych

- **Wewnątrz definicji klasy:**

```
class osoba {  
    public:  
        string imie;  
        string nazwisko;  
        int wiek;  
    osoba(string, string , int );  
    ~osoba(void);  
    void przedstawSie()  
    {  
        cout << "Nazywam sie " <<  
        imie << " " << nazwisko <<  
        endl;  
        cout << "mam " << wiek << "  
        lat." << endl;  
    };  
};
```

- **Na zewnątrz definicji klasy:**

```
class osoba {  
    public:  
        string imie;  
        string nazwisko;  
        int wiek;  
        void przedstawSie();//funkcja  
};  
void osoba::przedstawSie()  
{  
    cout << "Nazywam sie " << imie  
        << " " << nazwisko << endl;  
    cout << "mam " << wiek << "  
        lat." << endl;  
}
```



Definiowanie funkcji składowych

- Jeżeli funkcję składową zadeklarujemy wewnątrz definicji klasy to kompilator uzna, że chcemy aby była to funkcja *inline*,
- Definicja funkcji składowej poza definicją klasy powoduje, że funkcja nie jest automatycznie uznawana jako *inline*.

Definiowanie funkcji składowych

- Czy funkcja zdefiniowana poza klasą nie może już być typu inline?
- Może, ale trzeba to zaznaczyć słówkiem inline.

Zasłanianie nazw

- Nazwy składowych klasy mają zakres lokalny,
- W obrębie klasy zasłaniają elementy o tej samej nazwie leżące poza klasą,

Konstruktor i destruktor

- **Konstruktor** to specjalna funkcja składowa, która nazywa się tak samo jak klasa. W ciele konstruktora zamieszcza się instrukcje, które powodują ustawienie wartości początkowych przy tworzeniu obiektu danej klasy.
- **Destruktoorem** klasy jest funkcja poprzedzona ~, której zadaniem jest „posprząatanie” w pamięci po obiekcie, który został zniszczony. Destruktor jest wywoływany automatycznie.

Definiowanie konstruktora i destruktora

- **Konstruktor**

```
osoba::osoba(string i,string n, int w)
```

```
{  imie=i;  
   nazwisko=n;  
   wiek=w;  
};
```

- **Destruktor**

```
osoba::~~osoba(void)
```

```
{  
  cout<< "Obiekt został zniszczony" ;  
};
```


Domyślne konstruktory i destruktory

- Jeśli nie zadeklarujesz konstruktora lub destruktora, zrobi to za ciebie kompilator.
- Istnieje **wiele rodzajów konstruktorów**; niektóre z nich posiadają argumenty, inne nie. Konstruktor, którego można wywołać bez żadnych argumentów, jest nazywany konstruktorem domyślnym. Istnieje tylko **jeden rodzaj destruktora**. On także nie posiada argumentów.

O dziedziczeniu

- Dziedziczenie to specjalny mechanizm, który pozwala na tworzenie klas pochodnych względem klasy bazowej, czyli będących niejako rozbudowaną wersją klas już istniejących.
- Proces tworzenia klasy pochodnej jest bardzo prosty:

```
class Nowa_Klasa : public Nazwa_Klasy_Bazowej
```

- w tym przypadku **Nowa_Klasa** jest klasą pochodną i dziedziczy zawartość pojedynczej klasy bazowej

O dziedziczeniu

- Nie dziedziczymy:
 - Konstruktorów,
 - Destruktorów,
 - Operatorów przypisania.

O dziedziczeniu

- Kiedy tworzony jest obiekt klasy pochodnej najpierw wywoływany jest konstruktor klasy bazowej, a następnie konstruktor klasy pochodnej,
- Kiedy obiekt klasy pochodnej jest usuwany najpierw wywoływany jest destruktory klasy pochodnej , a potem destruktory klasy bazowej.

Przykład –bez dziedziczenia:

```
class pojazd
{
    public:
        int predkosc;
        int przyspieszenie;
        int ilosc_kol;
        int kolor;
};
```

```
class super_pojazd
{
    public:
        int predkosc;
        int przyspieszenie;
        int ilosc_kol;
        int kolor;
        int dopalacz;
};
```

Przykład ze strony <http://guidecpp.cal.pl/cplusplus/polimorph>

Przykład z dziedziczeniem

```
class pojazd
{
    public:
        int predkosc;
        int przyspieszenie;
        int ilosc_kol;
        int kolor;
};
```

```
class super_pojazd : public
    pojazd
{
    public:
        int dopalacz;
};
```

Sposoby dziedziczenia: public, protected, private

składniki w klasie podstawowej	sposób dziedziczenia	składniki w klasie pochodnej
prywatne chronione publiczne	prywatne	niedostępne niedostępne niedostępne
prywatne chronione publiczne	chronione	niedostępne chronione chronione
prywatne chronione publiczne	publiczne	niedostępne chronione publiczne

Przykład ze strony <http://guidecpp.cal.pl/cplus,polimorph>

Interfejs a implementacja

- Gdzie umieszczać deklaracje klasy i definicje metod ?
- Każda funkcja, którą zadeklarujesz dla klasy, musi posiadać definicję. Definicja jest nazywana także implementacją funkcji. Podobnie jak w przypadku innych funkcji, definicja metody klasy posiada nagłówek i ciało.

Interfejs a implementacja c.d.

- Deklaracja musi znajdować się w pliku, który może zostać znaleziony przez kompilator.
- W pliku, w którym umieszczasz deklarację funkcji, możesz umieścić również jej definicję, ale nie należy to do dobrych obyczajów.
- Zgodnie z konwencją zaadoptowaną przez większość programistów, deklaracje umieszcza się w tak zwanych plikach nagłówkowych, zwykle posiadających tę samą nazwę, lecz z rozszerzeniem *.h*, *.hp* lub *.hpp*.

Plik nagłówkowy

- Na przykład, deklarację klasy `Cat` powinieneś umieścić w pliku o nazwie `Cat.hpp`, zaś definicję metod tej klasy w pliku o nazwie `Cat.cpp`. Następnie powinieneś dołączyć do pliku `.cpp` plik nagłówkowy, poprzez umieszczenie na początku pliku `Cat.cpp` następującej dyrektywy:

```
#include "Cat.hpp"
```

- Informuje ona kompilator, by wstawił w tym miejscu zawartość pliku `Cat.hpp` tak, jakbyś ją wpisał ręcznie.
- Uwaga: w niektórych kompilatorach wielkość liter w nazwie pliku powinna zgadzać się z wielkością liter w nazwie pliku na dysku.

Plik nagłówkowy c.d.

- Dlaczego masz się trudzić, rozdzielając program na pliki `.hpp` i `.cpp`, skoro i tak plik `.hpp` jest wstawiany do pliku `.cpp`?
- W większości przypadków klient klasy nie dba o szczegóły jej implementacji. Odczytanie pliku nagłówkowego daje mu wystarczającą ilość informacji by zignorować plik implementacji. Poza tym, ten sam plik `.hpp` możesz dołączać do wielu różnych plików `.cpp`.



KONIEC