

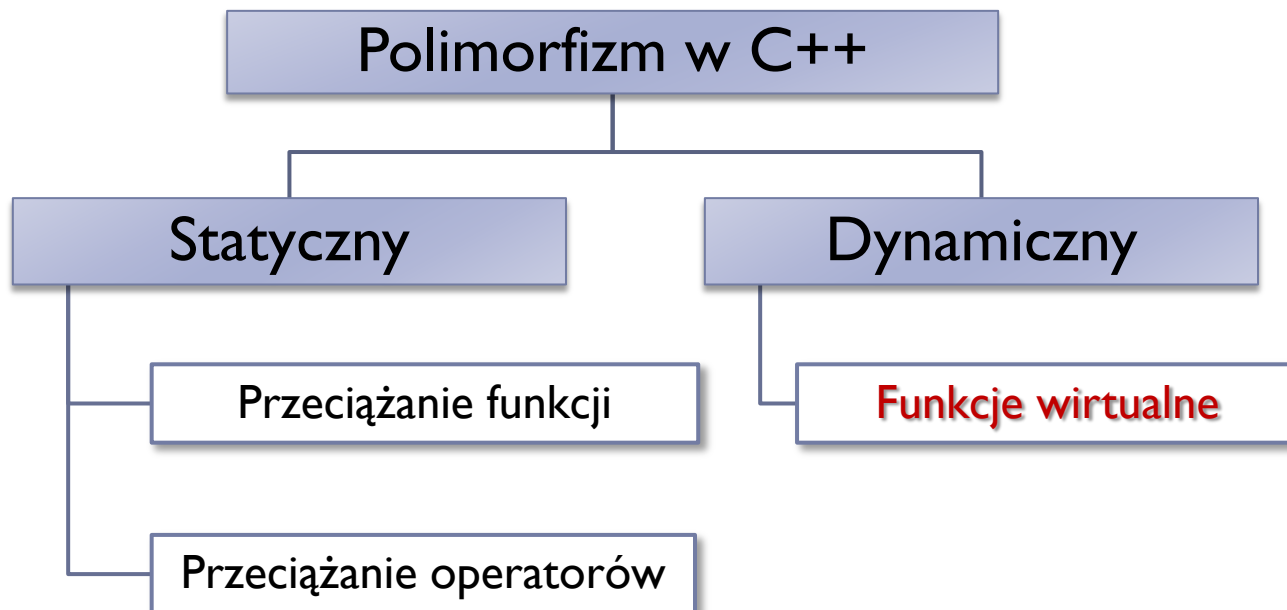
Wykład 9: Polimorfizm i klasy wirtualne



Polimorfizm i metody wirtualne



W programowaniu obiektowym **polimorfizm** (wielopostaciowość) to cecha umożliwiająca różne zachowanie tych samych metod wirtualnych (funkcji wirtualnych) w czasie wykonywania programu.



Wiązania statyczne i dynamiczne

Wiązania:

- deklaracja zmiennej powoduje związanie zmiennej z typem,
- wykonanie instrukcji podstawienia powoduje związanie zmiennej z (nową) wartością

Wiązania dzielimy na dwie klasy:

1. **Wiązania statyczne (wczesne wiązania)** czyli takie, które następują przed wykonaniem programu i nie zmieniają się w trakcie jego działania.
2. **Wiązania dynamiczne (późne wiązania)** to te, które następują lub zmieniają się w trakcie działania programu.

Wiązania statyczne i dynamiczne

1. Wiązania statyczne (wczesne)

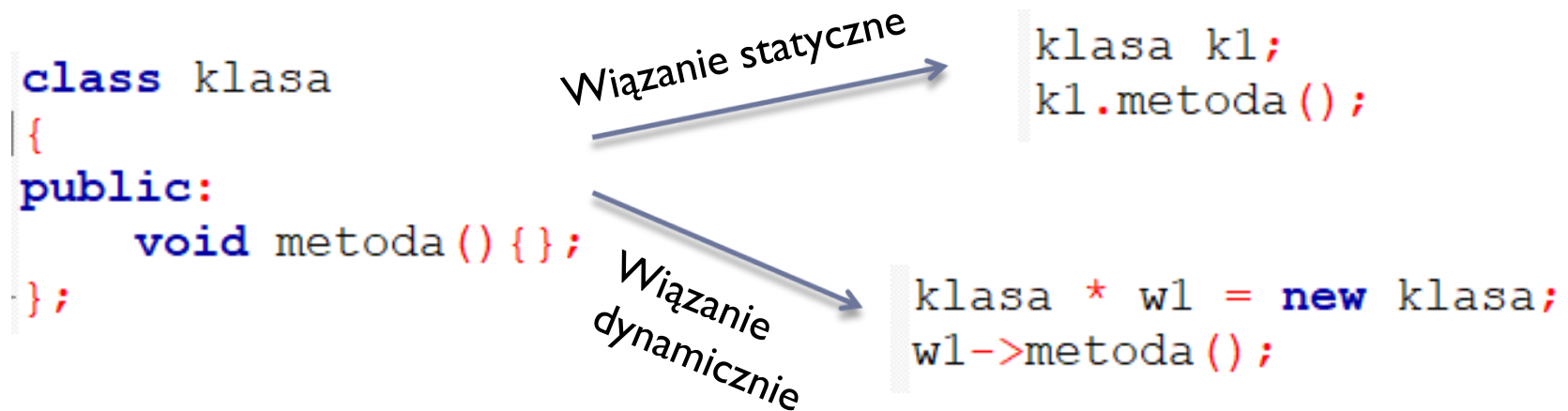
```
int zmienna;
```

2. Wiązania dynamiczne (późne)

```
int *wskaznik;  
wskaznik = &zmienna;
```

Wiązania statyczne i dynamiczne dla obiektów

- ✓ Obiekty zadeklarowane za pomocą zmiennych niewskaźnikowych są alokowane na stosie (jak zwykłe zmienne lokalne); wiązanie jest wówczas zawsze statyczne.
- ✓ Obiekty stworzone za pomocą wskaźnika i operatora **new** są powiązane dynamicznie





- ✓ W języku C++ możemy korzystać z **polimorfizmu** za pomocą **metod wirtualnych**. Dzięki niemu mamy pełną kontrolę nad wykonywanym programem, nie tylko w momencie kompilacji (wiązanie statyczne) ale także podczas działania programu (wiązanie dynamiczne) – niezależnie od różnych wyborów użytkownika.
- ✓ W C++ nie ma konieczności korzystania z polimorfizmu. Zostanie on automatycznie włączony podczas zadeklarowania przynajmniej jednej metody wirtualnej w danej klasie.

Polimorfizm

```
1  class Bazowa {
2      public:
3          int a;
4      };
5
6  class Pochodna : public Bazowa {
7      public:
8          int b;
9      };
10
11 int main()
12 {
13     Bazowa *bazowa = new Pochodna();
14     Pochodna *pochodna = new Pochodna();
15     return 0;
16 }
```

Typ statyczny obiektu

Typ dynamiczny obiektu

- ✓ Zmienna wskaźnikowa mająca typ pewnej klasy bazowej może wskazywać obiekty tejże klasy oraz klas pochodnych



```
1  #include <iostream>
2  using namespace std;
3
4  class Bazowa {
5  public:
6      void fun() { cout << "Bazowa \n"; }
7  };
8
9  class Pochodna : public Bazowa {
10 public:
11     void fun() { cout << "Pochodna \n"; }
12 };
13
14 int main()
15 {
16
17     Bazowa *bazowa = new Pochodna();
18     Pochodna *pochodna = new Pochodna();
19     bazowa->fun(); //wyswietli: bazowa
20     pochodna->fun(); //wyswietli: pochodna
21     return 0;
22 }
```

Bazowa
Pochodna



```
17   Bazowa *bazowa = new Pochodna ();
18   Pochodna *pochodna = new Pochodna ();
19   bazowa->fun (); //wyswietli: bazowa
20   pochodna->fun (); //wyswietli: pochodna
```

Bazowa
Pochodna

To, która metoda zostanie wywołana zależy od **typu wskaźnika** na obiekt. Jest to wspomniane wcześniej **wiązanie statyczne**. Kompilator już podczas kompilacji programu wie, jakiego **typu statycznego** są obiekty i jakie metody mają zostać wywołane.

Polimorfizm



```
1  #include <iostream>
2  using namespace std;
3
4  class Bazowa {
5  public:
6      void virtual fun() { cout << "Bazowa \n"; }
7  };
8
9  class Pochodna : public Bazowa {
10 public:
11     void fun() { cout << "Pochodna \n"; }
12 };
13
14 int main()
15 {
16
17     Bazowa *bazowa = new Pochodna();
18     Pochodna *pochodna = new Pochodna();
19     bazowa->fun(); //wyswietli: pochodna
20     pochodna->fun(); //wyswietli: pochodna
21     return 0;
22 }
```

Pochodna
Pochodna



Polimorfizm

```
4 class Bazowa {  
5     public:  
6         void virtual fun() { cout << "Bazowa \n"; }  
7     };
```

```
17     Bazowa *bazowa = new Pochodna();  
18     Pochodna *pochodna = new Pochodna();  
19     bazowa->fun(); //wyswietli: pochodna  
20     pochodna->fun(); //wyswietli: pochodna
```

Pochodna
Pochodna

Dzięki dodaniu do klasy bazowej **metod wirtualnych**, uruchomimy mechanizm **polimorfizmu**. Wczesne wiązanie statyczne nie będzie miało wtedy żadnego znaczenia, ponieważ to która funkcja zostanie wywołana będzie zależało od późnego wiązania dynamicznego.



```
class Bazowa {  
public:  
    void virtual fun ()=0;  
};
```

Metodę wirtualną (abstrakcyjną) deklaruje się za pomocą **virtual** i dodatkowego pseudopodstawienia **"=0"** (nie ma sensu pisać ciała metody jeżeli w żadnych okolicznościach kod ten nie może zostać wykonany)



- ✓ Zmienna wskaźnikowa mająca typ pewnej klasy bazowej może wskazywać obiekty tej klasy oraz klas pochodnych - a zatem jest polimorficzna.
- ✓ Zmienne niewskaźnikowe nie mogą być polimorficzne.
- ✓ Gdy używamy zmiennej polimorficznej do wywołania metody zdefiniowanej w jednej z klas pochodnych, wywołanie to musi zostać związane z właściwą definicją metody.
- ✓ Metody, które mają być wiązane dynamicznie, deklaruje się ze słowem kluczowym **virtual**.
- ✓ **virtual** oznacza, że dana metoda może być zredefiniowana w klasach pochodnych, a zatem jej wywołanie należy traktować jako polimorficzne.



Klasy abstrakcyjne

Klasy abstrakcyjne

- ✓ **Klasy abstrakcyjne to klasy dla których nie można stworzyć obiektu.**
- ✓ Klasy abstrakcyjne istnieją po to, aby z niej dziedziczyć.
- ✓ Często zdarza się że mamy kilka klas które mają pewną ilość cech wspólnych choć między nimi samymi nie zachodzi relacja dziedziczenia (żadna z klas nie jest szczególnym przypadkiem innej klasy). W takiej sytuacji można wydzielić bazową klasę gdzie zawarte były by wszystkie wspólne cechy.

Daje to szereg korzyści:

- jeśli postanowimy zmienić jakieś pole wspólne, to zmiany dokonujemy tylko w klasie bazowej,
- jeśli pojawią się nowe wspólne cechy które potrzebujemy to dodajemy je tylko w jednej klasie
- możliwość wywołań polimorficznych

Klasy abstrakcyjne

Aby klasa była abstrakcyjna to musi mieć przynajmniej jedną metodę czysto wirtualną - czyli metodę wirtualną która nie ma ciała.

```
class Ab  
{  
    public:  
        virtual void metodaCzystoWirtualna () =0;  
};
```

Próba utworzenia obiektu klasy abstrakcyjnej skończy się błędem kompilatora.



Klasy abstrakcyjne jako interfejsy

Interfejs to klasa abstrakcyjna która ma tylko i wyłącznie metody czysto wirtualne i nie ma żadnych pól.

Interfejsy są przydatne gdy chcemy zupełnie niezwiązanym ze sobą obiektom udostępnić taki sam zestaw metod. W ten sposób możemy napisać uniwersalne funkcje które wymagają od obiektów jedynie określonego zestawu metod,

W języku C++ interfejs może być zdefiniowany jako klasa abstrakcyjna



Literatura:

W prezentacji wykorzystano przykłady i fragmenty:

- Grębosz 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.
- Kernighan B.W., Ritchie D. M.: ***język ANSI C***, Wydawnictwo Naukowo Techniczne.

Dla bardziej zaawansowanych:

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

