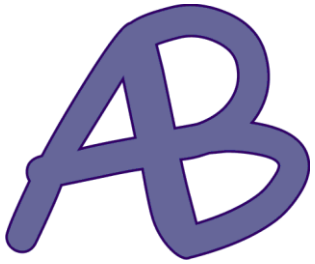
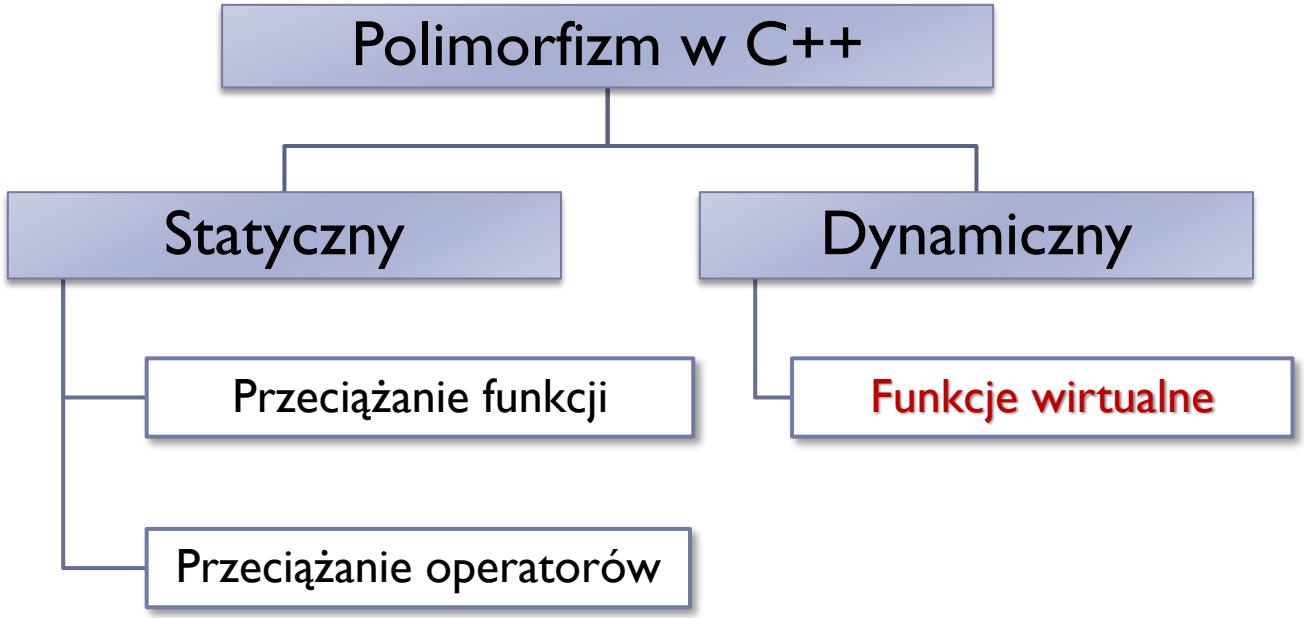


Wykład :
Polimorfizm i klasy wirtualne

Polimorfizm i metody wirtualne

Polimorfizm

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



Polimorfizm statyczny - kompilacyjny

Przeciążanie funkcji

```
class Wektor
{
public:
    double x, y;
    Wektor(double x = 0, double y = 0) : x(x), y(y) {}
    Wektor operator+(const Wektor &w) const
    {
        return Wektor(x + w.x, y + w.y);
    }
};
```

Przeciążanie operatorów

```
class Wektor
{
public:
    double x, y;
    Wektor(double x = 0, double y = 0) : x(x), y(y) {}
    Wektor operator+(const Wektor &w) const
    {
        return Wektor(x + w.x, y + w.y);
    }
};
```

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

```
class klasa
{
public:
    void metoda(){};
};
```

Wiązanie statyczne

```
klasa k1;
k1.metoda();
```

Wiązanie dynamicznie

```
klasa * w1 = new klasa;
w1->metoda();
```

Polimorfizm

- ✓ 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

```
class Bazowa {
public:
    int a;
};

class Pochodna : public Bazowa {
public:
    int b;
};

int main()
{
    Bazowa *bazowa = new Pochodna();
    Pochodna *pochodna = new Pochodna();
    return 0;
}
```

Typ statyczny obiektu

Typ dynamiczny obiektu

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

Polimorfizm

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

class Bazowa {
public:
    void fun() { cout << "Bazowa \n"; }
};

class Pochodna : public Bazowa {
public:
    void fun() { cout << "Pochodna \n"; }
};

int main()
{

    Bazowa *bazowa = new Pochodna();
    Pochodna *pochodna = new Pochodna();
    bazowa->fun();  //wyswietli: bazowa
    pochodna->fun();//wyswietli: pochodna
    return 0;
}
```

```
Bazowa
Pochodna
```

Polimorfizm

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

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

class Bazowa {
public:
    void virtual fun() { cout << "Bazowa \n"; }
};

class Pochodna : public Bazowa {
public:
    void fun() { cout << "Pochodna \n"; }
};

int main()
{

    Bazowa *bazowa = new Pochodna();
    Pochodna *pochodna = new Pochodna();
    bazowa->fun();  //wyswietli: pochodna
    pochodna->fun();//wyswietli: pochodna
    return 0;
}
```

```
Pochodna
Pochodna
```

Polimorfizm

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

```
Bazowa *bazowa = new Pochodna();
Pochodna *pochodna = new Pochodna();
bazowa->fun();   //wyswietli: pochodna
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.

Polimorfizm

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

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

Polimorfizm

- ✓ 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.

Tablica wskaźników do klas pochodnych

Tablica wskaźników do klas pochodnych

Jeżeli dwie klasy pochodne dziedziczą po wspólnej klasie bazowej i nadpisują jej metody wirtualne, możliwe jest **umieszczenie wskaźników do obiektów tych klas pochodnych w jednej tablicy wskaźników typu klasy bazowej**.

Dzięki temu możemy korzystać z polimorfizmu dynamicznego.

To klasyczne zastosowanie polimorfizmu dynamicznego — pozwala pisać kod **niezależny od konkretnych typów obiektów**, operując na wskaźnikach lub referencjach do klasy bazowej.

Tablica wskaźników do klas pochodnych

```
class Zwierze
{
public:
    virtual void dajGlos() const
    {
        cout << "Zwierzę wydaje dźwięk" << endl;
    }
    virtual ~Zwierze() {}
};
```

Klasa bazowa - abstrakcyjna

Klasy pochodne

```
class Pies : public Zwierze
{
public:
    void dajGlos() const override
    {
        cout << "Hau hau" << endl;
    }
};
```

```
class Kot : public Zwierze
{
public:
    void dajGlos() const override
    {
        cout << "Miau miau" << endl;
    }
};
```

Tablica wskaźników do klas pochodnych

- ✓ Każdy element tablicy to wskaźnik do klasy bazowej Zwierze.
- ✓ Dzięki metodom wirtualnym, przy wywołaniu dajGlos() zostanie wybrana wersja metody odpowiadająca rzeczywistemu typowi obiektu (czyli klasy Pies lub Kot).

```
int main()
{
    Zwierze *zoo[3];
    zoo[0] = new Zwierze();
    zoo[1] = new Pies();
    zoo[2] = new Kot();

    for (int i = 0; i < 3; ++i)
    {
        zoo[i]->dajGlos(); // dynamiczne wywołanie odpowiedniej wersji metody
    }

    for (int i = 0; i < 3; ++i)
        delete zoo[i];

    return 0;
}
```

Klasy wirtualne (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 Figura
{
public:
    virtual double pole() = 0;
    virtual double obwod() = 0;
};
```

Ta konstrukcja oznacza, że metoda jest czysto wirtualna to znaczy nie ma żadnego kodu (ciała)

Próba utworzenia obiektu klasy abstrakcyjnej skończy się błędem kompilatora.

Klasy abstrakcyjne

Na marginesie:

```
class Figura
{
public:
    virtual double pole() const = 0;
    virtual double obwod() const = 0;
};
```

W wielu przykładach pojawi się też Słowo kluczowe const. Oznacza ono, że metoda nie zmienia stanu obiektu, tzn. nie modyfikuje jego pól danych.

Klasy abstrakcyjne - przykład

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

// Klasa abstrakcyjna "osoba„ Służy jako ogólny interfejs dla klas pochodnych (np. student, wykładowca)
// Nie można utworzyć obiektu klasy abstrakcyjnej
class osoba
{
protected:
    string imie;
    string nazwisko;
    string uczelnia;
public:
    // Konstruktor z wartościami domyślnymi
    osoba(string Imie = "", string Nazwisko = "", string Uczelnia = "UTH Rad.")
        : imie(Imie), nazwisko(Nazwisko), uczelnia(Uczelnia) {}
    // Metody czysto wirtualne – wymuszają implementację w klasach pochodnych Dzięki temu klasa staje się
abstrakcyjna
    virtual void wczytajDane() = 0;
    virtual void wypiszDane() = 0;

    // Wirtualny destruktor umożliwia poprawne usuwanie obiektów klas pochodnych przez wskaźnik do klasy
bazowej
    virtual ~osoba() {}
};
```

Klasy abstrakcyjne

```
// Klasa pochodna "student", dziedziczy po klasie abstrakcyjnej "osoba"
class student : public osoba
{
protected:
    string kierunek;

public:
    // Konstruktor przekazujący dane do klasy bazowej oraz inicjalizujący pole "kierunek"
    student(string Imie = "", string Nazwisko = "", string Kierunek = "Informatyka", string Uczelnia = "UTH Rad.")
        : osoba(Imie, Nazwisko, Uczelnia), kierunek(Kierunek) {}

    // Implementacja metod czysto wirtualnych z klasy bazowej
    void wczytajDane() override;
    void wypiszDane() override;
};
```

Implementacje metod na następnym slajdzie

Klasy abstrakcyjne

Implementacje metod klasy student (Z poprzedniego slajdu)

```
void student::wczytajDane()
{
    // Przykład nadpisywania metody klasy bazowej – polimorfizm
    cout << "Imię: ";
    cin >> imie;
    cout << "Nazwisko: ";
    cin >> nazwisko;
    cout << "Kierunek: ";
    cin >> kierunek;
    cout << "Uczelnia: ";
    cin >> uczelnia;
}
void student::wypiszDane()
{
    cout << "Imię i nazwisko: " << imie << " " << nazwisko << endl;
    cout << "Kierunek: " << kierunek << endl;
    cout << "Uczelnia: " << uczelnia << endl;
}
```

Klasy abstrakcyjne

```
// Klasa pochodna "wykladowca", również dziedziczy po klasie abstrakcyjnej "osoba"
class wykladowca : public osoba
{
protected:
    string katedra;

public:
    wykladowca(string Imie = "", string Nazwisko = "", string Katedra = "Katedra Informatyki",
string Uczelnia = "UTH Rad.")
        : osoba(Imie, Nazwisko, Uczelnia), katedra(Katedra) {}

    // Implementacja metod czysto wirtualnych
    void wczytajDane() override;
    void wypiszDane() override;
};
```

Implementacje metod na następnym slajdzie

Klasy abstrakcyjne

Implementacje metod klasy wykladowca (Z poprzedniego slajdu)

```
void wykladowca::wczytajDane()
{
    cout << "Imię: ";
    cin >> imie;
    cout << "Nazwisko: ";
    cin >> nazwisko;
    cout << "Katedra: ";
    cin >> katedra;
    cout << "Uczelnia: ";
    cin >> uczelnia;
}

void wykladowca::wypiszDane()
{
    cout << "Imię i nazwisko: " << imie << " " << nazwisko << endl;
    cout << "Katedra: " << katedra << endl;
    cout << "Uczelnia: " << uczelnia << endl;
}
```

Klasy abstrakcyjne

```
int main()
{
    // Tablica wskaźników do klasy bazowej – przykład wykorzystania polimorfizmu
    osoba *tab[] = {
        new wykladowca("Artur", "Bartoszewski"),
        new student("Jan", "Kowalski"),
        new student("Anna", "Nowak")};

    // Dzięki wirtualnym metodom, wywoływane są odpowiednie implementacje klas pochodnych
    for (int i = 0; i < 3; i++)
        tab[i]->wypiszDane();

    // Zwalnianie pamięci – wywoływane są poprawnie destruktory klas pochodnych
    for (int i = 0; i < 3; i++)
        delete tab[i];

    return 0;
}
```

Interfejsy

Klasy abstrakcyjne jako interfejsy

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

- ✓ Używany do definiowania zbioru operacji, które muszą implementować klasy pochodne.

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 określonego zestawu metod.

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

Klasy abstrakcyjne jako interfejsy

```
class Pracownik
{
public:
    virtual void wypiszInfo() const = 0;
    virtual ~Pracownik() {}
};
```

Interfejs

Klasa implementująca Interfejs

```
class PracownikEtatowy : public Pracownik
{
    string nazwisko;

public:
    PracownikEtatowy(const string &n) : nazwisko(n) {}
    void wypiszInfo() const override
    {
        cout << "Pracownik etatowy: " << nazwisko << endl;
    }
};
```

Metody i klasy finalne

Metody i klasy finalne

Słowo kluczowe final może zostać użyte w deklaracji metody WIRTUALNEJ w klasie bazowej, aby zablokować możliwość jej dalszego przesłaniania (override) w klasach pochodnych.

Zastosowanie:

- Wymusza ostateczność implementacji metody.
- Zapobiega błędom w projektowaniu hierarchii dziedziczenia.

Metody i klasy finalne

```
class Zwierze
{
public:
    virtuaL void dajGlos() final
    {
        cout << "Zwierzę wydaje dźwięk" << endl;
    }
};
```

```
class Pies : public Zwierze
{
public:
    // Błąd kompilacji! Nie można przesłonić metody oznaczonej jako final
    void dajGlos()     {
        cout << "Hau hau" << endl;
    }
};
```

Metody i klasy finalne

Można oznaczyć całą klasę jako final.
Uniemożliwia dalsze dziedziczenie.

```
class Ostateczna final
{
    // brak możliwości utworzenia klasy pochodnej
};
```

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