Język C++

część 6

polimorfizm i funkcje wirtualne

Jarosław Gramacki
Instytut Informatyki i Elektroniki

- Polimorfizm i funkcje wirtualne - wprowadzenie

  Polimorfizm:

  – ma zastosowanie przy pracy z hierarchiami klas, gdy chcemy traktować obiekt NIE jako reprezentanta specjalizowanego typu, lecz jako obiekt typu bazowego

  – polimorfizm (wielopostaciowość) pozwala w jednolity sposób traktować obiekty klas z hierarchii dziedziczenia przy zachowaniu ich charakterystycznego zachowania

  – polimorfizm to cecha, która oznacza, że ta sama instrukcja w kodzie źródłowym programu może w czasie działania programu wywoływać różne funkcje

  – … a wszystko to dzięki prostej zdolności stosowania tej samej nazwy metody w więcej niż jednej klasie w hierarchii dziedziczenia

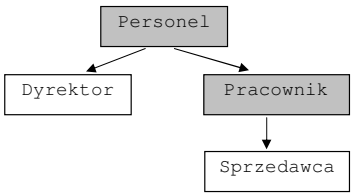
  – można nie stosować polimorfizmu, ale …

  – mechanizm korzystający np. z dyrektyw `switch`, choć możliwy do zastosowania w praktyce, jest wysoce nieskuteczny ! i mało kto z niego korzysta !

- Polimorfizm i funkcje wirtualne - wprowadzenie

Funkcje wirtualne:

- mechanizm funkcji wirtualnych umożliwia korzystanie z polimorfizmu w praktyce
- funkcje wirtualne to takie funkcje, które używane są identycznie w całej hierarchii klas.
- od strony technicznej działanie mechanizmu funkcji wirtualnych sprowadza się to do tzw. późnego wiązania (ang. late binding) metod przy ich wywoływaniu (wybór metody do wykonania następuje na podstawie analizy obiektu "na którym stoimy")
- Przykład:
  dla hierarchii `Personel, Pracownik, Sprzedawca, Dyrektor` utwórzmy tablicę na dane pracowników firmy, w której będziemy przechowywać dane osób zatrudnionych w firmie



- Polimorfizm i funkcje wirtualne - wprowadzenie

Polimorfizm czasu przebiegu:

- Przeładowywanie metod umożliwia ujednolicenie działań wykonywanych na danych różnych typów i jest najbardziej przydatne podczas tworzenia małych klas, gdyż ma charakter statyczny (trzeba przewidzieć wszystkie typy danych na których trzeba będzie wykonywać działania).
- Czasami potrzebne jest rozwiązanie przewidujące możliwość zmiany implementacji danej metody w podklasie; wówczas jej wersja zdefiniowana w nadklasie może w ogóle nie wykonywać żadnego działania. Tego typu polimorfizm nazywa się polimorfizmem czasu przebiegu (ang. *runtime polymorphism*).

- Funkcje wirtualne i polimorfizm - przykład

```
class TablePersonel {              // "inteligentna" tablica pracowników
public:      TablePersonel( int _rozmiar ); ~TablePersonel();
             void add( Personel* p );
             void print() const;
             float suma_wyplat() const;
private:     Personel **table;
             int rozmiar;
             int indeks;  };

TablePersonel::TablePersonel( int _rozmiar )
                  : rozmiar( _rozmiar ), indeks( 0 )
     { table = new Personel* [ rozmiar ]; }

TablePersonel::~TablePersonel() {
    delete [] table; }

void TablePersonel::add( Personel* p ) {
    if( indeks < rozmiar )
      table[ indeks++ ] = p;  }

void TablePersonel::print() const {
    for( int i = 0; i < indeks; i++ )
      table[ i ] -> print();  }

float TablePersonel::suma_wyplat() const {
    for( int i = 0; i < indeks; i++ )
      total += table[ i ] -> oblicz_place();     return total;  }
```

wskaźnik do wskaźnika do **Personel, czyli ...**

... wskaźnik do klasy **bazowej**

konwersja do wskaźnika klasy pochodnej nastąpi automatycznie. Patrz `main()` następna strona

problematyczna instrukcja !

problematyczna instrukcja !

- Funkcje wirtualne i polimorfizm - przykład

```
void main() {
    TablePersonel tp( 100 );

    Pracownik e1( "Kowalski", 1, 70.0, 169.0 );
    Sprzedawca v1( "Malinowski", 2, 95.0, 169.0, 0.05, 10000 );
    Dyrektor d1( "Kuczma", 3, 15000, 1500, 25 );

    tp.add( &e1 );
    tp.add( &v1 );
    tp.add( &d1 );
    tp.print();

    cout << "Wyplata pracownika #1 = " << e1.oblicz_place() << endl;
    cout << "Wyplata pracownika #2 = " << v1.oblicz_place() << endl;
    cout << "Wyplata pracownika #3 = " << d1.oblicz_place() << endl;

    cout << "Suma wyplat = " << tp.suma_wyplat() << endl;  }
```

- problem: wykonanie **`table[ i ]->print();`** wyświetli tylko nazwisko i numer telefonu osoby (nie uwzględnione zostanie, że o np. `Sprzedawcy` posiadamy więcej informacji
- wykonanie **`total += table[ i ]->oblicz_place();`** w ogóle zakończy się błędem informującym, że nie znaleziono funkcji `Personel::oblicz_place()`
- powód: nie jest możliwe określenie, która funkcja powinna być wywołana dla konkretnego obiektu

- Funkcje wirtualne i polimorfizm - przykład

– prostszy przykład, który (na razie) nie używa klasy `TablePersonel`
(lepiej nadaje się do analizy samego mechanizmu funkcji wirtualnych)
– używany mechanizm: VMT (ang. virtual method table), tablice wskaźników v-tables

pomijamy to zagadnienie

```
void main()
{
    Pracownik e( "Kowalski", 1, 70.0, 169.0 );
    Sprzedawca v( "Malinowski", 2, 95.0, 169.0, 0.05, 10000 );

    Personel* ptr;  // wskaźnik do klasy bazowej

    ptr = &e;
    ptr -> print(); // wywołanie przez wskaźnik do klasy bazowej

    ptr -> &v;
    ptr -> print(); // j.w.
}
```

charakter błędów, które się pojawią jest analogiczny jak poprzednio

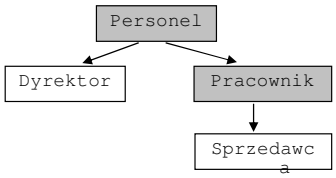
- Funkcje wirtualne i polimorfizm - przykład

– c.d z poprzedniego slajdu

– uwaga: gdy kompilator jest w stanie "rozwiązać" wywołanie funkcji w czasie kompilacji, czyni to
– zachodzi to w sytuacji, gdy funkcję wywołujemy za pośrednictwem pewnego obiektu klasy, a nie wskaźnika do tego obiektu (tak jak w przykładzie powyżej). Mówimy wtedy o tzw. realizacji statycznej wywołania.

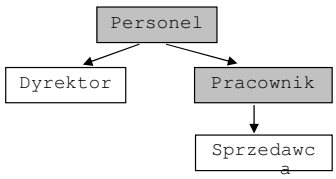
```
Pracownik e( "Kowalski", 1, 70.0, 169.0 );
e.print();                                    // Pracownik::print()


Sprzedawca v( "Malinowski", 2, 95.0, 169.0, 0.05, 10000 );
v.print();                                    // Sprzedawca:print()
```

- Funkcje wirtualne i polimorfizm - analiza przykładu
  - zasygnalizowany problem generalnie występuje w przypadku, gdy wywołujemy dowolną funkcję w hierarchii klas za pomocą wskaźnika do klasy bazowej
  - problem w naszym przypadku polega na tym, że każde wywołanie funkcji za pośrednictwem `table[i]` wywołuje funkcję z klasy `Personel`. W rzeczywistości chcemy aby została wywołana funkcja właściwa dla typu obiektu, na który wskazuje wskaźnik `table[i]` (wskaźnik do klasy bazowej)
  - kompilator nie jest w stanie tego określić, gdyż dopiero w trakcie wykonywania programu staje się wiadome jakie wskaźniki (do jakich obiektów) pojawią się jako elementy tablicy `table`
  - gdy np. `table[2]` jest typu `Sprzedawca`, to należy wywołać `Sprzedawca::print()`
    gdy np. `table[5]` jest typu `Dyrektor`, to należy wywołać `Dyrektor::print()`



- Funkcje wirtualne i polimorfizm - analiza przykładu
  - rozwiązaniem problemu jest zastosowanie mechanizmu funkcji wirtualnych
  - jeżeli następuje wywołanie funkcji wirtualnej za pomocą wskaźnika do klasy bazowej, to wybór funkcji do wykonania uzależniony jest od typu obiektu wskazywanego przez wskaźnik

to jest w zasadzie meritum polimorfizmu !

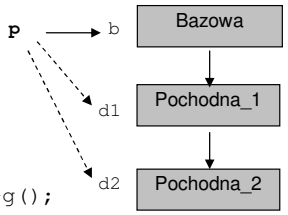


- Funkcje wirtualne i polimorfizm - ogólny przykład

```
class Bazowa {
public:
    virtual void f() { cout << "f() : klasa Bazowa !" << endl; }
    void g()         { cout << "g() : klasa Bazowa !" << endl; }
};

class Pochodna_1 : public Bazowa {
public:
    virtual void f() { cout << "f() : klasa Pochodna 1 !" << endl; }
    void g()         { cout << "g() : klasa Pochodna 1 !" << endl; }
};

class Pochodna_2 : public Pochodna_1 {
public:
    virtual void f() { cout << "f() : klasa Pochodna 2 !" << endl; }
    void g()         { cout << "g() : klasa Pochodna 2 !" << endl; }
};

void main() {
    Bazowa b;
    Pochodna_1 d1;
    Pochodna_2 d2;

    Bazowa *p = &b;  p->f();  p->g();
    p = &d1;         p->f();  p->g();
    p = &d2;         p->f();  p->g();
}
```

Funkcja pozostaje wirtualna od tego miejsca w dół całej hierarchii dziedziczenia

słowo `virtual` konieczne tylko w klasie bazowej. Powtarzanie go w klasach pochodnych jest nieobowiązkowe, choć zalecane




```
Wynik:

f(): klasa Bazowa !
g(): klasa Bazowa !
f(): Pochodna 1 !
g(): klasa Bazowa !
f(): Pochodna 2 !
g(): klasa Bazowa !
```

posługujemy się wskaźnikiem `p` do klasy bazowej

- Funkcje wirtualne i polimorfizm - ogólny przykład
  - komentarz:
  - jedyna konieczna zmiana to dodanie słowa `virtual` przy odpowiednich funkcjach w klasie bazowej
  - ogólna zasada: w klasie bazowej powinno się używać funkcji wirtualnych, gdy przewiduje się przedefiniowanie tych funkcji w klasach pochodnych
  - uwaga:
    - w klasach pochodnych nie ma obowiązku powtarzania słowa virtual przy przedefiniowanych funkcjach. Umieszczenie go jednak zwiększa czytelność kodu
    - w klasie (klasach) pochodnej nie ma obowiązku redefiniowania wirtualnej funkcji z klasy bazowej. Gdy jej nie będzie, to wywołana zostanie po prostu funkcja z klasy bazowej (czyli tak jak w przypadku zwykłych funkcji - nie wirtualnych)

- Funkcje wirtualne i polimorfizm - zastosowanie w klasie Personel
  - aby klasa `TablePersonel` mogła funkcjonować poprawnie, konieczne są zmiany w klasie `Personel`
  - uwaga:
    funkcja `oblicz_place()` w klasie bazowej jest tylko "oparciem" dla słowa `virtual`. Inaczej pojawi się błąd przy kompilacji linii
    "`total += table[ i ]->oblicz_place();`"
  - Po tych zmianach, program testujący klasę `TablePersonel` będzie działał poprawnie

```
class Personel
{
public:
    Personel( const char* _nazwisko, short _biuro = 0 );
    ~Personel() { delete [] nazwisko; }
    virtual void print() const;
    virtual float oblicz_place() const { return 0; }
    void set_biuro( short _biuro ) { biuro = _biuro; }
    const char* get_telefon() const;
private:
    char* nazwisko;
    short biuro;
};
```

metoda „atrapa”

- Funkcje wirtualne i polimorfizm - destruktory wirtualne
  - z reguły jeśli klasa definiuje co najmniej jedną funkcję wirtualną, należy też zdefiniować destruktor wirtualny
    (nawet gdyby musiał on być nic-nie-wykonującą atrapą)

```
class Personel
{
public:
    Personel( const char* _nazwisko, short _biuro = 0 );
    virtual ~Personel() { delete [] nazwisko; }
    virtual void print() const;
    virtual float oblicz_place() const { return 0; }
    void set_biuro( short _biuro ) { biuro = _biuro; }
    const char* get_telefon() const;
private:
    char* nazwisko;
    short biuro;
};
```

tu akurat ma on konkretny cel

- Funkcje wirtualne i polimorfizm - destruktory wirtualne

  - stosowanie polimorfizmu prowadzi zazwyczaj do sytuacji, w której większość obiektów przetwarzana jest z pomocą wskaźnika do ich klasy bazowej
  - załóżmy, że utworzono dynamicznie (operatorem `new`) obiekty różnych typów, a następnie próbujemy zwolnić zajmowaną przez nie pamięć korzystając z operatora `delete`
  - użycie `delete` wobec wskaźnika do klasy bazowej (bo założyliśmy, patrz wyżej, że do obiektu dostajemy się korzystając ze wskaźnika do klasy bazowej) spowoduje wykonanie destruktora klasy bazowej a NIE (jakbyśmy oczekiwali) destruktora klasy pochodnej !

- Funkcje wirtualne i polimorfizm - destruktory wirtualne

```
class Bazowa {
public:
    virtual void f();
    // brak destruktora wirtualnego  };

class Pochodna : public Bazowa {
public:
    Pochodna(int _rozmiar);
    ~Pochodna();
private:
    int *pi;
};

Pochodna::Pochodna(int _rozmiar) {
  pi = new int[ _rozmiar];
}

Pochodna:~Pochodna() {
  delete [] pi;
}

void main()
 {
  Bazowa *pb;             // wskaźnik do klasy bazowej
  pb = new Pochodna(10);
  delete pb;              // NIE wykona się destruktor klasy pochodnej
 }
```

UWAGA!
funkcje języka C takie jak: malloc() i free() w żaden sposób nie zapewniają opisanej tu funkcjonalności – gdy powstawał język C, NIE było przecież obiektów !

- Funkcje wirtualne i polimorfizm - destruktory wirtualne

  - `delete pb;` zwalnia pamięć przydzieloną przez obiekt Pochodna, czyli przez zmienna `int*`
  - pamięć przydzielona przez `Pochodna:Pochodna()` NIE została zwolniona (czyli `10 int`)
  - aby poprawić błąd, należy w klasie `Bazowa` zdefiniować destruktor wirtualny
  - po tej czynności, instrukcja `delete pb;` spowoduje wykonanie destruktora obiektu wskazywanego przez `pb` czyli destruktora `Pochodna::~Pochodna()`

- Funkcje wirtualne i polimorfizm - destruktory wirtualne

```
class Bazowa
{
public:
    virtual ~Bazowa() {}              // dodano tylko to
    virtual void f();
    // brak destruktora wirtualnego
};

class Pochodna : public Bazowa {
public:
    Pochodna(int _rozmiar);
    ~Pochodna();
private:
    int *pi;  };

Pochodna::Pochodna(int _rozmiar) {
  pi = new int[ _rozmiar]; }

Pochodna:~Pochodna() {
  delete [] pi; )

void main()
 {
  Bazowa *pi;
  pb = new Pochodna(10);
  delete pb;            // teraz WYKONA SIĘ destruktor klasy pochodnej
 }
```

- Funkcje wirtualne i polimorfizm - destruktory wirtualne

  – jeśli więc jakaś klasa definiuje funkcje wirtualne, to powinna definiować też wirtualny destruktor (nawet w postaci "atrapy" jak wyżej, nie wykonującej w odnośnej klasie żadnych czynności)

  – zasady tej należy przestrzegać, gdyż w chwili pisania klasy bazowej nie można założyć, że kiedyś nie zostanie napisana klasa pochodna, wymagająca wywołania destruktora

  – Uwaga: **konstruktorów** nigdy nie poprzedza się słowem `virtual`.

  – Nie ma takiej potrzeby, gdyż tworząc nowy obiekt, zawsze podajemy typ tegoż obiektu (tak jak deklarując zmienna prostą zawsze musimy podać jej typ)

```
int zmienna;
?   zmienna; // BLAD
```