15 Polimorfizm

Polimorfizm (ang. *polymorphism*) oznacza wielopostaciowość. W szczególności dotyczy ona funkcji członkowskich (metod) definiowanych w klasach. W języku C++ można implementować dwa rodzaje polimorfizmu:

- polimorfizm statyczny,
- polimorfizm dynamiczny.

15.1. Polimorfizm statyczny

Polimorfizm statyczny (ang. *static polymorphism*) zachodzi w czasie kompilacji programu. Dlatego też często nazywa się go **polimorfizmem w czasie kompilacji** (ang. *compile-time polymorphism*).

Stosowanie polimorfizmu statycznego może być związane:

- z przeciążaniem metod,
- z przesłanianiem metod.

15.1.1. Przeciążanie metod

Polimorfizm statyczny (ang. *static polymorphism*) może wynikać z wykorzystania mechanizmu **przeciążania metod** (ang. *methods overloading*) zdefiniowanych w obrębie danej klasy. Mianowicie jeśli w danej klasie zostanie zdefiniowanych wiele metod o tej samej nazwie, ale różniących się od siebie liczbą i/lub typem parametrów (ewentualnie typem zwracanej wartości), to mamy do czynienia z przeciążeniem tych metod.

Wywołanie przeciążonej metody powoduje, że kompilator już na etapie kompilacji programu porównuje to wywołanie z definicjami wszystkich przeciążonych metod zawartych w klasie. Mówiąc dokładniej, kompilator porównuje liczbę i typy argumentów wywołania funkcji z liczbą i typami parametrów w poszczególnych definicjach funkcji przeciążonych. To samo dotyczy typu wartości zwracanej przez metody. Na tej podstawie kompilator wybiera najbardziej odpowiednią funkcję do wywołania.

Ze względu na to, że działania kompilatora w opisanej sytuacji są realizowane na etapie kompilacji programu, polimorfizm statyczny często jest nazywany **polimorfizmem w czasie kompilacji** (ang. *compile-time polymorphism*).

Wielu programistów polimorfizm statyczny utożsamia z tzw. **wczesnym wiązaniem** (ang. *early binding*). Przy czym wspomniane **wiązanie** (ang. *binding*) należy rozumieć jako powiązanie wywołania metody z treścią (definicją) metody. Ujmując to dokładniej, wiązanie oznacza skojarzenie nazwy (identyfikatora) wywoływanej metody z adresem bloku pamięci operacyjnej, w którym ta metoda jest przechowywana. Technicznie rzecz biorąc, skutkuje to zastąpieniem wywołania metody instrukcją języka maszynowego wywołującą skok w pamięci do adresu odpowiedniej metody.

UWAGA

Wczesne wiązanie często jest nazywane również **wiązaniem statycznym** (ang. *static binding*) lub **wiązaniem w czasie kompilacji** (ang. *compile-time binding*).

Przykład 15.1

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

// Definicja klasy Pracownik:
class Pracownik {
public:
    // Deklaracje zmiennych członkowskich:
    string imie, nazwisko, stanowisko;
    // Deklaracje (prototypy) przeciążonych metod zwrocDane():
    string zwrocDane();
    void zwrocDane(string&, string&, string&);
    /* UWAGA
     * Funkcje członkowskie zadeklarowane powyżej mają taką samą nazwę, ale różnią się parametrami
     * (liczbą i/lub typem) i/lub typem zwracanej wartości.
     */
};
// Definicje funkcji członkowskich zwrocDane() z klasy Pracownik:
string Pracownik::zwrocDane() {
    return imie + " " + nazwisko + ", stanowisko: " + stanowisko;
}
void Pracownik::zwrocDane(string &pImie, string &pNazwisko,
                          string &pStanowisko) {
    pImie = imie;
    pNazwisko = nazwisko;
    pStanowisko = stanowisko;
}
```

```
int main() {
    // Utworzenie obiektu pracownik jako instancji klasy Pracownik:
    Pracownik pracownik;
    // Nadanie wartości zmiennym członkowskim obiektu pracownik:
    pracownik.imie = "Jan";
    pracownik.nazwisko = "Kowalski";
    pracownik.stanowisko = "nauczyciel";

    // Wywołanie przeciążonej funkcji członkowskiej (metody) zwrocDane():
    cout << "Dane pracownika: " << pracownik.zwrocDane() << endl;
    /* UWAGA
     * Kompilator już na etapie kompilacji zdecyduje, która z funkcji zwrocDane() zdefiniowanych w klasie Pracownik
     * zostanie w tym miejscu wywołana.
     */
    string imie, nazwisko, stanowisko; // zmienne pomocnicze
    // Wywołanie przeciążonej funkcji członkowskiej zwrocDane():
    pracownik.zwrocDane(imie, nazwisko, stanowisko);
    /* UWAGA
     * Kompilator już na etapie kompilacji zdecyduje, która ze zdefiniowanych funkcji zwrocDane() zostanie
     * w tym miejscu wywołana.
     */
    cout << "Dane pracownika: ";
    cout << imie + " " + nazwisko + ", stanowisko: " + stanowisko << endl;

    return 0;
}
```

W programie zdefiniowano klasę `Pracownik`, która zawiera m.in. definicje przeciążonych metod o nazwie `zwrocDane`. Pomimo że metody te mają identyczną nazwę, różnią się zarówno liczbą, jak i typem parametrów, a ponadto typami wartości zwracanych.

W programie głównym metoda `zwrocDane()` została wywołana dwukrotnie. Kompilator już na etapie kompilacji programu porównuje postacie tych wywołań z definicjami metod, a w szczególności argumenty tych wywołań z parametrami formalnymi, jak też typy zwracanych przez nie wartości. Po przeprowadzeniu analizy kompilator wybiera tę postać metody `zwrocDane()`, która najbardziej pasuje do danego wywołania.

Mamy tutaj do czynienia z wielopostaciowością metody `zwrocDane()`. Wielopostaciowość ta ma miejsce w czasie kompilacji programu.

Podsumowując — w zaprezentowanym programie zachodzi polimorfizm statyczny.

Ćwiczenie 15.1

Zmodyfikuj program z przykładu 15.1 — zaimplementuj w klasie `Pracownik` hermetyzację (ukrycie) danych.

15.1.2. Przesłanianie metod

Stosowanie polimorfizmu statycznego w programowaniu obiektowym może być również związane z mechanizmem **przesłaniania metod** (ang. *method overriding*) w ramach łańcucha dziedziczenia klas (ang. *class inheritance chain*).

Najprostszy sposób implementacji efektu przesłaniania metod polega na tym, że zarówno w klasie bazowej, jak i w jej klasie pochodnej definiuje się funkcje członkowskie (metody) o tych samych nazwach. Także liczba i typy parametrów formalnych oraz typy wartości zwracanych przez te metody są takie same.

Jeśli założyć, że wywołanie jednej z metod o opisanych powyżej cechach jest skojarzone w sposób bezpośredni z obiektem będącym instancją klasy pochodnej, tj. przy wykorzystaniu identyfikatora tego obiektu oraz operatora dostępu ., metoda zdefiniowana w klasie pochodnej przesłania (ang. *overrides*) metodę o tej samej nazwie zdefiniowaną w jej klasie bazowej. Patrząc na to samo zjawisko z drugiej strony — metoda zdefiniowana w klasie bazowej zostaje przesłonięta (ang. *overridden*) przez metodę o tej samej nazwie zdefiniowaną w jej klasie pochodnej.

Dopasowanie odpowiedniej definicji metody do postaci jej wywołania (a dokładniej: do wyrażenia zawierającego jej wywołanie) polega w opisywanej sytuacji na podjęciu decyzji, która metoda ma zostać wykonana — czy ta zdefiniowana w klasie bazowej, czy też metoda o tej samej nazwie zdefiniowana w klasie pochodnej. Proces ten jest realizowany w całości przed wykonaniem programu — w czasie jego kompilacji. Tym samym mamy tu do czynienia z **wczesnym wiązaniem** (ang. *early binding*) wywołania metody z jej treścią (definicją). To zaś oznacza, że zachodzi polimorfizm statyczny, tj. polimorfizm w czasie kompilacji (ang. *compile-time polymorphism*).

UWAGA

Efekt przesłaniania metod występuje również w polimorfizmie dynamicznym, związanym z wykorzystaniem metod wirtualnych. Tematyka ta została omówiona szczegółowo w podrozdziale 15.2.2.

Przykład 15.2

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

// Definicja klasy bazowej Pracownik:
class Pracownik {
public:
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};
```

```
// Definicja klasy pochodnej Nauczyciel:
class Nauczyciel: public Pracownik {
public:
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};
// Definicja klasy pochodnej Wychowawca:
class Wychowawca: public Nauczyciel {
public:
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};
/* UWAGA
 * W każdej z klas wchodzących w skład łańcucha dziedziczenia zadeklarowano metodę publiczną o takiej samej
 * nazwie oraz identycznych parametrach (liczbie i typie) i typach wartości zwracanych.
 */
// Definicje metod:
void Pracownik::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Pracownik"
         << endl;
}
void Nauczyciel::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Nauczyciel"
         << endl;
}
void Wychowawca::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Wychowawca"
         << endl;
}
/* UWAGA
 * Implementacja i funkcjonalność każdej ze zdefiniowanych metod zwrocDane() jest w ogólności dowolna.
 */

int main() {
    // Utworzenie obiektu pracownik1 jako instancji klasy Pracownik:
    Pracownik pracownik1;
    // Wywołanie metody zwrocDane():
    pracownik1.zwrocDane();
    /* UWAGA
     * Powyższe wywołanie dotyczy metody zwrocDane() zdefiniowanej w klasie bazowej Pracownik.
     */

    // Utworzenie obiektu pracownik2 jako instancji klasy Nauczyciel:
    Nauczyciel pracownik2;
    // Wywołanie metody zwrocDane():
```

```
    pracownik2.zwrocDane();
    /* UWAGA
     * Powyższe wywołanie dotyczy metody zwrocDane() zdefiniowanej w klasie pochodnej Nauczyciel.
     */

    // Utworzenie obiektu pracownik3 jako instancji klasy Wychowawca:
    Wychowawca pracownik3;
    // Wywołanie metody zwrocDane():
    pracownik3.zwrocDane(); // zostanie wywołana metoda zdefiniowana w klasie pochodnej Wychowawca

    return 0;
}
```

W programie zdefiniowano trzy klasy: `Pracownik`, `Nauczyciel` i `Wychowawca`, pomiędzy którymi określono relacje dziedziczenia. W każdej z wymienionych klas zdefiniowano metodę o takiej samej nazwie — `zwrocDane`. Zarówno liczba, jak i typy parametrów metod `zwrocDane()` zdefiniowanych w każdej z wymienionych powyżej klas są takie same — tj. brak parametrów. To samo dotyczy wartości zwracanych przez te metody na zewnątrz — brak wartości zwracanych (`void`).

Metoda `zwrocDane()` została wywołana w programie trzykrotnie. Pierwsze ze wspomnianych wywołań dotyczy obiektu `pracownik1` jako instancji klasy bazowej `Pracownik`. Drugie wywołanie jest związane z obiektem `pracownik2` należącym do klasy pochodnej `Nauczyciel`, a trzecie — z obiektem `pracownik3` klasy `Wychowawca`. W każdym z wymienionych przypadków wywoływana jest metoda zdefiniowana w klasie, której instancję stanowi powiązany z nią obiekt. Zatem wywołanie `pracownik1.zwrocDane()` powoduje wykonanie metody zdefiniowanej w klasie bazowej `Pracownik`, wywołanie `pracownik2.zwrocDane()` — wykonanie metody z klasy pochodnej `Nauczyciel`, a wywołanie `pracownik2.zwrocDane()` — wykonanie metody z klasy pochodnej `Wychowawca`.

Mamy tutaj do czynienia z efektem przesłaniania metod w łańcuchu dziedziczenia. Metoda `zwrocDane()` zdefiniowana w klasie `Wychowawca` przesłania metodę o tej samej nazwie zdefiniowaną w jej klasie bazowej, tj. klasie `Nauczyciel`. Z kolei metoda `zwrocDane()` zdefiniowana w klasie `Nauczyciel` przesłania metodę `zwrocDane()` z jej klasy bazowej, czyli z klasy `Pracownik`.

Można w tym przypadku mówić o wczesnym wiązaniu (ang. *early binding*) wywołań metody `zwrocDane()` z jej definicjami w poszczególnych klasach wchodzących w skład łańcucha dziedziczenia. Proces ten zachodzi w czasie kompilacji programu. To z kolei oznacza, że w odniesieniu do wymienionej metody zachodzi polimorfizm statyczny, tj. polimorfizm w czasie kompilacji (ang. *compile-time polymorphism*). Metoda `zwrocDane()` jest wielopostaciowa (na etapie kompilacji programu).

Ćwiczenie 15.2

Zmodyfikuj program z przykładu 15.2 — uzupełnij klasy `Pracownik`, `Nauczyciel` i `Wychowawca` o odpowiednie zmienne członkowskie (dane) i konstruktory. Metody `zwrocDane()` zdefiniowane w każdej z klas łańcucha dziedziczenia uzupełnij o kod pozwalający na wyświetlenie na ekranie monitora danych obiektu należącego do danej klasy.

15.2. Polimorfizm dynamiczny

Polimorfizm dynamiczny (ang. *dynamic polymorphism*) to polimorfizm, który zachodzi w trakcie wykonywania programu — już po zakończeniu procesu jego kompilacji. Dlatego też często jest on nazywany **polimorfizmem w czasie wykonywania programu** (ang. *runtime polymorphism*).

Jak wspomniano w poprzednim podrozdziale, zjawisko polimorfizmu w programowaniu obiektowym jest utożsamiane ze sposobem wiązania (ang. *binding*) wywołania metody z jej definicją (treścią). Biorąc pod uwagę czas, w którym ten proces jest realizowany, termin **późne wiązanie** (ang. *late binding*) oznacza, że zachodzi on w czasie wykonywania programu — w odróżnieniu od wiązania wczesnego, które następuje na etapie kompilacji programu.

UWAGA

Późne wiązanie wywołania metody z jej definicją jest również nazywane **wiązaniem w czasie wykonywania programu** (ang. *runtime binding*) lub **wiązaniem dynamicznym** (ang. *dynamic binding*).

Stosowanie polimorfizmu dynamicznego wiąże się z wykorzystaniem obiektów należących do różnych klas wchodzących w skład łańcucha dziedziczenia, do których dostęp został zrealizowany za pomocą wskaźników.

15.2.1. Wskaźniki w mechanizmie dziedziczenia

Dostęp do obiektów, które są instancjami klas wchodzących w skład łańcucha dziedziczenia, można zrealizować za pomocą wskaźników. Dotyczy to zarówno obiektów należących do klasy bazowej (lub klas bazowych), jak i obiektów klas pochodnych.

UWAGA

Tematyka wskaźników do obiektów — bez uwzględnienia mechanizmu dziedziczenia — została omówiona w podrozdziale 11.7.

Interesująca sytuacja występuje wówczas, gdy w celu uzyskania dostępu do obiektów, które wchodzą w skład łańcucha dziedziczenia, wykorzystuje się wskaźnik należący do typu

statycznego określonego przez klasę bazową. **Typ statyczny wskaźnika** (ang. *static type of pointer*, *pointer static type*) określa się w jego deklaracji. Na przykład deklaracja `Pracownik *w_pracownik;` oznacza, że typem statycznym wskaźnika `w_pracownik` jest klasa `Pracownik`.

Za pośrednictwem takiego wskaźnika można również uzyskać dostęp nie tylko do obiektów należących do klasy bazowej, ale też do obiektów będących instancjami jej klas pochodnych. Wynika to z jednej z kluczowych cech programowania obiektowego w języku C++, mianowicie z tego, że typ wskaźnika na obiekt klasy bazowej jest kompatybilny (czyli zgodny pod względem typu) z typem wskaźnika na obiekt klasy pochodnej.

Przykład 15.3

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

// Definicja klasy bazowej Pracownik:
class Pracownik {
public:
    string imie, nazwisko;
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};
// Definicja klasy pochodnej Nauczyciel:
class Nauczyciel: public Pracownik {
public:
    string przedmiot;
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};
// Definicja klasy pochodnej Wychowawca:
class Wychowawca: public Nauczyciel {
public:
    string klasa;
    // Prototyp metody publicznej zwrocDane():
    void zwrocDane();
};
/* UWAGA
 * W każdej z klas wchodzących w skład łańcucha dziedziczenia zadeklarowano metodę zwrocDane() —
 * o takiej samej nazwie oraz identycznych parametrach (liczbie i typie) i typie zwracanej wartości.
 */
// Definicje metod instancyjnych zwrocDane():
void Pracownik::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Pracownik"
         << endl;
    cout << "Imię: " << imie << endl;
```

```
    cout << "Nazwisko: " << nazwisko << endl;
}
void Nauczyciel::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Nauczyciel"
         << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
    cout << "Przedmiot: " << przedmiot << endl;
}
void Wychowawca::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Wychowawca"
         << endl;
    cout << "Imię: " << imie << endl;
    cout << "Nazwisko: " << nazwisko << endl;
    cout << "Przedmiot: " << przedmiot << endl;
    cout << "Klasa:" << klasa << endl;
}
/* UWAGA
 * Implementacja (i co za tym idzie — funkcjonalność) każdej ze zdefiniowanych metod zwrocDane() jest dowolna.
 */

int main() {
    // Deklaracja wskaźnika w_pracownik typu statycznego Pracownik:
    Pracownik *w_pracownik;
    /* UWAGA
     * Zmienna w_pracownik jest zmienną statyczną, która może wskazywać (z definicji) na obiekty typu bazowego
     * Pracownik oraz obiekty klas pochodnych klasy Pracownik, czyli obiekty klas Nauczyciel i Wychowawca.
     */
    // Utworzenie obiektu pracownik1 jako instancji klasy Pracownik:
    Pracownik pracownik1;
    // Przypisanie wskaźnikowi w_pracownik adresu obiektu pracownik1:
    w_pracownik = &pracownik1;
    // Nadanie wartości zmiennym członkowskim obiektu pracownik1:
    w_pracownik->imie = "Jan";
    w_pracownik->nazwisko = "Kowalski";
    // Wywołanie metody zwrocDane():
    w_pracownik->zwrocDane(); // zostaje wywołana metoda z klasy bazowej Pracownik
    cout << endl;

    // Utworzenie obiektu pracownik2 jako instancji klasy Nauczyciel:
    Nauczyciel pracownik2;
    w_pracownik = &pracownik2;
    w_pracownik->imie = "Adam";
    w_pracownik->nazwisko = "Nowak";
```

```
    /* UWAGA
     * Użycie dereferencji w_pracownik-> pozwala jedynie uzyskać dostęp do elementów członkowskich klasy
     * Nauczyciel odziedziczonych po klasie bazowej Pracownik — czyli do zmiennych imie i nazwisko.
     * Dostęp do elementów członkowskich zdefiniowanych bezpośrednio w klasie pochodnej Nauczyciel nie jest
     * możliwy. Tym samym próba wykonania instrukcji przypisania: w_pracownik->przedmiot = „Informatyka";
     * zakończy się komunikatem o błędzie i informacją, że klasa Pracownik nie zawiera elementu członkowskiego
     * przedmiot.
     */
    // Wywołanie metody zwrocDane():
    w_pracownik->zwrocDane(); // zostaje wywołana metoda odziedziczona po klasie bazowej Pracownik
    cout << endl;

    // Utworzenie obiektu pracownik3 jako instancji klasy Wychowawca:
    Wychowawca pracownik3;
    w_pracownik = &pracownik3;
    w_pracownik->imie = "Jan";
    w_pracownik->nazwisko = "Polski";
    // w_pracownik->przedmiot = „Informatyka"; INSTRUKCJA BŁĘDNA!
    // w_pracownik->klasa = „3A"; INSTRUKCJA BŁĘDNA!

    // Wywołanie metody zwrocDane():
    w_pracownik->zwrocDane(); // zostaje wywołana metoda odziedziczona po klasie bazowej Pracownik

    return 0;
}
```

Program zawarty w rozpatrywanym przykładzie to modyfikacja programu z przykładu 15.2. Modyfikacja polega na tym, że dostęp do obiektów: `pracownik1`, `pracownik2` i `pracownik3` uzyskano za pomocą jednego wskaźnika, `w_pracownik`, oraz operatora dostępu `->`.

W szczególności dostęp do obiektów: `pracownik1`, `pracownik2` i `pracownik3`, będących instancjami klas, odpowiednio, `Pracownik`, `Nauczyciel`, `Wychowawca`, które wchodzą w skład łańcucha dziedziczenia, uzyskano za pomocą jednego wskaźnika, `w_pracownik`. Wskaźnik `w_pracownik` został zdefiniowany jako `Pracownik *w_pracownik`. Tym samym typem statycznym wskaźnika `w_pracownik` jest klasa bazowa `Pracownik`. Wskaźnik `w_pracownik` może wskazywać (z definicji) zarówno na obiekty należące do klasy bazowej `Pracownik`, jak i na obiekty jej klas pochodnych, czyli `Nauczyciel` i `Wychowawca`.

Metoda `zwrocDane()`, zdefiniowana w każdej z klas wchodzących w skład łańcucha dziedziczenia, została wywołana w programie trzykrotnie. Pierwsze ze wspomnianych wywołań dotyczy obiektu `pracownik1`, będącego instancją klasy bazowej `Pracownik`. Drugie wywołanie jest związane z obiektem `pracownik2`, należącym do klasy pochodnej `Nauczyciel`, a trzecie — z obiektem `pracownik3` klasy `Wychowawca`. W każdym z wymienionych przypadków dostęp do danego obiektu zrealizowano za pomocą wskaźnika `w_pracownik` typu statycznego `Pracownik` oraz operatora strzałkowego `->`, czyli dereferencji `w_pracownik->`.

Skutkiem każdego ze wspomnianych wywołań jest wykonanie metody `zwrocDane()` zdefiniowanej w klasie bazowej `Pracownik` (a nie w klasach pochodnych), czyli wywołanie

`Pracownik::zwrocDane()`. Dzieje się tak dlatego, że wskaźnik `w_pracownik` stanowi (z definicji) referencję do typu bazowego `Pracownik`. Tym samym za jego pośrednictwem można uzyskać dostęp jedynie do elementów członkowskich klas pochodnych odziedziczonych po klasie bazowej — tj. odziedziczonych zmiennych członkowskich `imie` i `nazwisko` oraz odziedziczonej metody `zwrocDane()`. Dostęp do elementów członkowskich zdefiniowanych bezpośrednio w klasach pochodnych `Nauczyciel` i `Wychowawca` za pomocą dereferencji `w_pracownik->` nie jest możliwy.

Biorąc pod uwagę powyższe, w programie nie występuje przesłanianie metody `zwrocDane()`. Metoda ta jest metodą jednopostaciową, a nie wielopostaciową — jeśli spojrzeć z perspektywy jej wywołania za pośrednictwem wskaźnika `w_pracownik`. Tym samym żaden polimorfizm tutaj nie zachodzi.

Ćwiczenie 15.3

Zmodyfikuj program z przykładu 15.3 — zamiast dziedziczenia pojedynczego wielopoziomowego zastosuj dziedziczenie wielokrotne. W tym celu uzupełnij łańcuch dziedziczenia o zdefiniowane samodzielnie klasy `Przedmiot` i `Klasa`, zawierające odpowiednie zmienne członkowskie i/lub metody, oraz zmień relacje dziedziczenia pomiędzy klasami. Funkcjonalność zmodyfikowanego programu powinna być taka sama jak funkcjonalność programu z przykładu 15.3.

15.2.2. Metody wirtualne

Metody wirtualne (ang. *virtual methods*) to jedno z najważniejszych narzędzi polimorfizmu, czyli wielopostaciowości. Użycie metod wirtualnych jest związane z **polimorfizmem dynamicznym** (ang. *dynamic polymorphism*), a więc polimorfizmem zachodzącym **w czasie wykonywania** (ang. *runtime*) programu.

Metoda wirtualna to funkcja członkowska zdefiniowana w klasie bazowej, której deklaracja jest poprzedzona słowem kluczowym `virtual`. Zadeklarowanie metody w klasie bazowej jako wirtualnej pozwala na jej późniejsze „przedefiniowanie" (ang. *redefining*) w klasach pochodnych. Przy tym zachowane zostają charakterystyczne właściwości jej wywołań za pośrednictwem referencji do klasy bazowej.

W praktyce wspomniane powyżej odwołania są realizowane za pomocą wskaźnika należącego do typu statycznego klasy bazowej. Jednakże **typ dynamiczny wskaźnika** (ang. *dynamic type of pointer*, *pointer dynamic type*), który określa typ obiektu faktycznie wskazywanego przez wskaźnik, może się zmieniać w trakcie wykonywania programu.

UWAGA

Powiązanie wskaźnika z jego typem dynamicznym jest nazywane wiązaniem dynamicznym (ang. *dynamic linkage*) lub późnym wiązaniem (ang. *late binding*).

Wybór, która z metod zostanie wykonana jako skutek wywołania za pośrednictwem referencji do klasy bazowej — czy metoda zdefiniowana w klasie bazowej, czy któraś z metod o tej samej nazwie z klasy pochodnej — zależy od tego, czy metoda zdefiniowana w klasie bazowej została zadeklarowana jako wirtualna.

Stosowanie metod wirtualnych w programowaniu obiektowym jest ściśle powiązane z mechanizmem dziedziczenia. Uproszczone założenia umożliwiające wykorzystanie polimorfizmu dynamicznego wynikającego z użycia metod wirtualnych są następujące:

- W klasie bazowej zdefiniowano metodę wirtualną o określonych: nazwie, parametrach oraz typie wartości zwracanej.
- W klasach pochodnych zdefiniowano metody o nazwie identycznej z nazwą metody wirtualnej w klasie bazowej oraz takich samych parametrach (liczbie i typie) i typach zwracanych.
- Wywołania metod (o takiej samej nazwie jak nazwa funkcji wirtualnej w klasie bazowej) dotyczą obiektów będących instancjami różnych klas, które wchodzą w skład łańcucha dziedziczenia.
- Dostęp do obiektów należących do klas bazowej i pochodnych, połączonych ze sobą relacją dziedziczenia, jest realizowany przy użyciu wskaźnika należącego do typu statycznego klasy bazowej. Jednocześnie typ dynamiczny tego wskaźnika może się zmieniać w trakcie wykonywania programu.

Biorąc pod uwagę przedstawione powyżej założenia, wybór metody do wykonania zależy od typu dynamicznego wskaźnika, za którego pośrednictwem jest realizowany dostęp do danego obiektu. W rezultacie wybrana zostanie metoda zdefiniowana w klasie zgodnej z typem dynamicznym przetwarzanego obiektu. Metoda wirtualna z klasy bazowej zostaje wówczas **przesłonięta** (ang. *overridden*) przez metodę zdefiniowaną w tej klasie pochodnej, na której instancję (obiekt) faktycznie wskazuje wskaźnik.

Przykład 15.4

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

// Definicja klasy bazowej Pracownik:
class Pracownik {
public:
    // Prototyp metody wirtualnej zwrocDane():
    virtual void zwrocDane();
};
// Definicja klasy pochodnej Nauczyciel:
class Nauczyciel: public Pracownik {
public:
    // Prototyp metody zwrocDane():
```

```
    void zwrocDane();
};
// Definicja klasy pochodnej Wychowawca:
class Wychowawca: public Nauczyciel {
public:
    // Prototyp metody zwrocDane():
    void zwrocDane();
};
/* UWAGA
 * W każdej z klas wchodzących w skład łańcucha dziedziczenia zadeklarowano metodę publiczną o takiej samej nazwie
 * oraz identycznych parametrach (liczbie i typie) i typie zwracanej wartości.
 * Przy tym metoda zdefiniowana w klasie bazowej Pracownik jest metodą wirtualną.
 */
// Definicje metod zadeklarowanych w klasach Pracownik, Nauczyciel i Wychowawca:
void Pracownik::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Pracownik"
         << endl;
}
void Nauczyciel::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Nauczyciel"
         << endl;
}
void Wychowawca::zwrocDane() {
    cout << "Wywołanie metody zwrocDane() zdefiniowanej w klasie Wychowawca"
         << endl;
}

int main() {
    // Deklaracja wskaźnika w_pracownik typu statycznego Pracownik:
    Pracownik *w_pracownik;
    // Utworzenie obiektu pracownik1 jako instancji klasy Pracownik:
    Pracownik pracownik1;
    // Przypisanie wskaźnikowi w_pracownik adresu obiektu pracownik1:
    w_pracownik = &pracownik1;
    // Wywołanie metody zwrocDane():
    w_pracownik->zwrocDane(); // Zostanie wywołana metoda z klasy bazowej Pracownik.

    // Utworzenie obiektu pracownik2 jako instancji klasy Nauczyciel:
    Nauczyciel pracownik2;
    w_pracownik = &pracownik2;
    /* UWAGA
     * Realizacja przypisania w_pracownik = &pracownik2; powoduje zmianę typu dynamicznego wskaźnika
     * w_pracownik z typu Pracownik na typ Nauczyciel. Po wykonaniu przypisania wskaźnik ten wskazuje
     * na obiekt pracownik2 należący do typu Nauczyciel.
     */
```

```
    // Wywołanie metody zwrocDane():
    w_pracownik->zwrocDane();
    /* UWAGA
     * Ze względu na to, że typem dynamicznym wskaźnika w_pracownik jest typ Nauczyciel, wywołana
     * zostaje metoda zwrocDane() zdefiniowana w klasie pochodnej Nauczyciel.
     * Tym samym metoda wirtualna zwrocDane() zdefiniowana w klasie bazowej została przesłonięta przez metodę
     * zwrocDane() z klasy pochodnej.
     */
    // Utworzenie obiektu pracownik3 jako instancji klasy Wychowawca:
    Wychowawca pracownik3;
    w_pracownik = &pracownik3; // zmiana typu dynamicznego wskaźnika w_pracownik na typ
                               // Wychowawca
    // Wywołanie metody zwrocDane():
    w_pracownik->zwrocDane(); // zostanie wywołana metoda z klasy pochodnej Wychowawca

    return 0;
}
```

W klasie bazowej `Pracownik` zdefiniowano metodę wirtualną `zwrocDane()`. W klasach pochodnych `Nauczyciel` i `Wychowawca` wchodzących w skład łańcucha dziedziczenia zdefiniowano metody o identycznych nazwach, liczbach i typach parametrów oraz typach zwracanych.

Wskaźnik `w_pracownik` należy do typu statycznego `Pracownik`, co wynika z jego deklaracji: `Pracownik *w_pracownik;`. Można go zatem używać do wskazywania zarówno na obiekty należące do klasy bazowej `Pracownik`, jak i na obiekty będące instancjami klas pochodnych `Nauczyciel` i `Wychowawca`.

Typ dynamiczny wskaźnika `w_pracownik` — czyli typ obiektu, na który faktycznie wskazuje ten wskaźnik — zmienia się w trakcie wykonywania programu. Po wykonaniu przypisania `w_pracownik = &pracownik1;` typ dynamiczny wskaźnika `w_pracownik` zostaje ustalony na typ `Pracownik`, ponieważ obiekt `pracownik1` jest instancją klasy `Pracownik`. Na skutek zaś wykonania instrukcji przypisania, tj. `w_pracownik = &pracownik2;`, typ dynamiczny wskaźnika `w_pracownik` zmienia się na typ `Nauczyciel`. Wynika to z faktu, że obiekt `pracownik2` należy do klasy `Nauczyciel`. Ostatnia zmiana typu dynamicznego wskaźnika `w_pracownik` następuje po wykonaniu instrukcji `w_pracownik = &pracownik3;`. Wówczas typem dynamicznym wskaźnika `w_pracownik` jest `Wychowawca`.

Wszystkie wywołania metody `zwrocDane()` są realizowane za pośrednictwem wskaźnika `w_pracownik` — niezależnie od typu obiektu wskazywanego przez ten wskaźnik — za pomocą dereferencji `w_pracownik->`.

Jeśli wskaźnik `w_pracownik` wskazuje na obiekt `pracownik1`, będący instancją klasy `Pracownik`, wywoływana jest metoda `zwrocDane()` zdefiniowana w klasie `Pracownik`. Dzieje się tak dlatego, że typem dynamicznym wskaźnika `w_pracownik` jest klasa `Pracownik`.

Jeśli wskaźnik `w_pracownik` wskazuje na obiekt `pracownik2`, będący instancją klasy pochodnej `Nauczyciel`, wywoływana jest metoda `zwrocDane()` zdefiniowana w klasie `Nauczyciel` —

zgodnie z typem dynamicznym wskaźnika, którym jest typ `Nauczyciel`. Tym samym metoda `zwrocDane()` zdefiniowana w klasie pochodnej `Nauczyciel` przesłoniła metodę wirtualną o tej samej nazwie z klasy bazowej `Pracownik`.

Analogiczna sytuacja występuje po zmianie typu dynamicznego wskaźnika `w_pracownik` na typ `Wychowawca`. W tym przypadku wykonywana jest metoda `zwrocDane()` zdefiniowana w klasie pochodnej `Wychowawca`.

Biorąc pod uwagę powyższe rozważania, w programie zachodzi polimorfizm dynamiczny, który dotyczy metod `zwrocDane()` zdefiniowanych w klasie bazowej i klasach pochodnych łańcucha dziedziczenia. Wspomniany polimorfizm został uzyskany dzięki zadeklarowaniu funkcji `zwrocDane()` w klasie bazowej jako wirtualnej. W programie występują wiązania dynamiczne, czyli wiązania typu dynamicznego wskaźnika `w_pracownik` z typem obiektu, na który faktycznie wskazuje ten wskaźnik.

Ćwiczenie 15.4

Zmodyfikuj program zawarty w przykładzie 15.4 — zamiast obiektów: `pracownik1`, `pracownik2` i `pracownik3`, dla których pamięć operacyjna została przydzielona statycznie na stosie, wykorzystaj obiekty, dla których pamięć operacyjna została zaalokowana w sposób dynamiczny na stercie.

15.3. Pytania i zadania kontrolne

15.3.1. Pytania

1. Na czym polega zjawisko polimorfizmu statycznego?
2. Wyjaśnij znaczenie pojęcia przeciążania metod.
3. Na czym polega polimorfizm dynamiczny? Wymień najważniejsze cechy polimorfizmu dynamicznego.
4. Omów zagadnienie przesłaniania metod.
5. Co to jest metoda wirtualna?
6. Jakie są zalety stosowania polimorfizmu statycznego i dynamicznego?

15.3.2. Zadania

1. Napisz program pozwalający przetwarzać dane uczniów. Uwzględnij następujące dane: imię, nazwisko, numer w dzienniku, klasę, grupę. Wykorzystaj mechanizm polimorfizmu statycznego w odniesieniu do metody, której zadaniem jest wyświetlenie na ekranie monitora wszystkich lub wybranych danych ucznia. Wykonaj testy działania programu na danych kilku uczniów. Dane uczniów zainicjuj w treści programu.
2. Napisz program pozwalający przetwarzać wybrane dane smartfona: markę, model, rok produkcji, cenę, system operacyjny, przekątną wyświetlacza. Zastosuj mechanizm poli-

morfizmu statycznego do metod, których zadaniem jest realizacja operacji wejścia/wyjścia. Zapewnij możliwość wprowadzenia wszystkich lub wybranych danych smartfona z klawiatury oraz wyświetlenia wszystkich lub wybranych danych na ekranie monitora. Wykonaj testy działania programu na danych kilku smartfonów.

3. Napisz program pozwalający przetwarzać dane pracowników szpitala, a w szczególności lekarzy i ordynatorów oddziałów szpitalnych. Wykorzystaj mechanizm dziedziczenia pojedynczego wielopoziomowego oraz polimorfizm dynamiczny. Załóż, że każdy lekarz ma określoną specjalizację (np. kardiologia, chirurgia), a każdy ordynator jest lekarzem i jednocześnie kieruje pojedynczym oddziałem w szpitalu (np. oddziałem wewnętrznym, dziecięcym). Dostęp do obiektów odpowiadających przykładowym lekarzom i ordynatorom zrealizuj w tradycyjny sposób, przy użyciu operatora dostępu . (kropki). Wykonaj testy działania programu na danych kilku lekarzy i ordynatorów.
4. Zrób tak jak w zadaniu 3., z tym że dostęp do obiektów odpowiadających przykładowym lekarzom i ordynatorom zrealizuj za pośrednictwem wskaźnika należącego do typu statycznego zgodnego z zaproponowanym przez siebie typem bazowym (np. klasą `Pracownik`) przy użyciu operatora strzałkowego `->`.
5. Napisz program pozwalający przetwarzać dane wybranych pracowników na komendzie policji. Uwzględnij policjantów oraz naczelników wydziałów. Wykorzystaj mechanizm dziedziczenia wielokrotnego oraz polimorfizm dynamiczny. Załóż, że każdy policjant ma określony stopień służbowy (np. podkomisarz, komisarz, inspektor), a każdy naczelnik jest policjantem i jednocześnie kieruje jednym wydziałem na komendzie (np. wydziałem ruchu drogowego, wydziałem prewencji). Dostęp do obiektów odpowiadających przykładowym policjantom i naczelnikom zrealizuj w tradycyjny sposób, przy użyciu operatora dostępu . (kropki). Wykonaj testy działania programu na danych kilku policjantów i naczelników wydziałów.
6. Zrób tak jak w zadaniu 5., z tym że dostęp do obiektów odpowiadających przykładowym policjantom i naczelnikom zrealizuj za pośrednictwem wskaźników należących do typów statycznych zgodnych z zaproponowanymi przez siebie typami bazowymi (np. klasami: `Pracownik`, `Stopien`, `Wydzial`) z zastosowaniem operatora strzałkowego `->`.