Instrukcja III

Wstęp

Na tych zajęciach zaznajomimy się z dziedziczeniem oraz dynamicznym (wirtualnym) polimorfizmem. Są to podstawowe narzędzia programowania obiektowego, którym zawdzięcza ono w dużej mierze swoją popularność. Pozwalają one na pisanie przejrzystego kodu i konstruowanie łatwych w użyciu interfejsów. Dodajmy jednak, że współcześnie w C++ odchodzi się od tych metod na rzecz statycznych (tzn. rozstrzyganych w czasie kompilacji) rozwiązań. Powodem tej zmiany jest ich koszt w wydajności programu. Warto je jednak znać, szczególnie, że w wielu przypadkach czytelność kodu może być ważniejsza od kilku nanosekund czasu wykonania.

Dziedziczenie

Dziedziczenie jest prostym, ale bardzo potężnym konceptem. Sprowadza się do następującej zasady: jeżeli klasa `D` dziedziczy po klasie `B`, to `D` otrzymuje "w spadku" funkcjonalność (pola i metody) `B`, którą może dodatkowo rozszerzyć. Mówiąc ściślej, obiekt typu `D` zawiera w sobie podobiekt klasy `B`. Pozwala nam to na uproszczenie struktury kodu poprzez komponowanie nowych klas z już istniejących. Na przykład, jeżeli mamy klasy `Ołówek` i `Gumka`, to chcąc zaimplementować klasę reprezentującą ołówek z gumką na końcu, możemy po prostu stworzyć pustą klasę, która dziedziczy po `Ołówek` i `Gumka`.

```
class Ołówek { /* ... */ };
class Gumka  { /* ... */ };

class OłówekZGumką : public Ołówek, public Gumka {};
```

Tego typu strategię możemy znaleźć w bibliotece I/O. Klasy strumieni dziedziczą po odpowiednich klasach bazowych w zależności od tego czy mają być strumieniami wejścia, wyjścia, czy obu.

Inne możliwe zastosowanie dziedziczenia polega na użyciu klasy bazowej jako wspólnego interfejsu dla wielu klas pochodnych. Zobaczmy, jak może to wyglądać w praktyce.

Zadanie 1 Napisz klasę `Figura`, która posiada prywatną zmienną `double pole`.

Zadanie 2 Napisz setter i getter dla pola tej klasy.

Zadanie 3 Napisz klasy `Kolo` i `Kwadrat`, które dziedziczą po `Figura`.

Zadanie 4 Dodaj do napisanych klas konstruktory, które liczą pole figury na podstawie podanych wymiarów geometrycznych.

Dzięki dziedziczeniu po klasie `Figura` omijamy wielokrotne definiowanie pola `pole` oraz settera i gettera. Nie trudno wyobrazić sobie, że dla większych klas taki zabieg pozwala zaoszczędzić wiele linijek kodu.

Modyfikatory dostępu

W przykładzie powyżej, w konstruktorach klas pochodnych zmuszeni byliśmy używać settera pola klasy bazowej, gdyż było ono prywatne. Wygodniej nam jednak operować bezpośrednio na zmiennej. Z drugiej strony, nie chcemy wystawiać tej zmiennej bezpośrednio do użytkownika. Rozwiązaniem tego dylematu jest trzeci (i ostatni) specyfikator dostępu w C++: `protected` (chroniony). Pola chronione mogą być czytane i modyfikowane przez metody klas, które dziedziczą po klasie, do której chronione pole należy, ale nie przez żadne inne.

Zadanie 5 Uczyń pole `pole` chronionym. Usuń setter tego pola i zmodyfikuj odpowiednio konstruktory klas `Kwadrat` i `Kolo`.

Dodatkowo możemy przy dziedziczeniu zmodyfikować dostęp do pól i metod klasy bazowej za pomocą modyfikatorów dostępu. Nie będziemy tego ćwiczyć, ale dla kompletności bieżącego tekstu podajemy niżej wygodną "ściągawkę".

Modyfikator dostępu	Dostęp w klasie bazowej	Dostęp w klasie pochodnej
`public`		
	`public`	`public`
	`protected`	`protected`
	`private`	brak dostępu
`protected`		
	`public`	`protected`
	`protected`	`protected`
	`private`	brak dostępu
`private`		

Modyfikator dostępu	Dostęp w klasie bazowej	Dostęp w klasie pochodnej
	`public`	`private`
	`protected`	`private`
	`private`	brak dostępu

Konstruktory klas pochodnych

Fakt, że klasa pochodna zawiera w sobie podobiekt klasy bazowej ma konsekwencje dla sposobu jej konstrukcji. Przypomnijmy, że dla niedziedziczących klas tworzenie obiektu przebiego wg. następującego schematu:

1. Następuje alokacja pamięci na obiekt (na stosie lub na stercie, dla przebiegu procesu konstrukcji nie jest to istotne)
2. Wołany jest konstruktor
3. Pola obiektu są inicjalizowane przy pomocy listy inicjalizacyjnej i/lub konstruktorów domyślnych tych pól
4. Wykonywane jest ciało konstruktora

Zobaczmy, co stanie się, gdy spróbujemy bezpośrednio przenieść ten schemat na klasy dziedziczące.

Zadanie 6 Zmodyfikuj konstruktory klas `Kolo` i `Kwadrat` tak, aby inicjalizowały pole `pole` w swojej liście inicjalizacyjnej. Czy kod się skompiluje?

Błąd kompilacji w powyższym zadaniu wynika z tego, że - z punktu widzenia konstrukcji - klasa bazowa zachowuje się jak pole klasy pochodnej. Ogólny schemat tworzenia obiektu wygląda więc nieco inaczej, niż ten przedstawiony powyżej:

1. Następuje alokacja pamięci na obiekt
2. Wołany jest konstruktor
3. Obiekty klas bazowych konstruowane są za pomocą wskazanych konstruktorów (lub, jeżeli takowe nie zostały wskazane, konstruktorów domyślnych)
4. Pola obiektu są inicjalizowane przy pomocy listy inicjalizacyjnej i/lub konstruktorów domyślnych tych pól
5. Wykonywane jest ciało konstruktora

W kodzie wygląda to następująco:

```
class B1 { /* ... */ };
class B2 { /* ... */ };
// ...
class D : public B1, public B2 // ...
{
    D(/* ... */) : B1{/* ... */}, B2{/* ... */} // ...
    {
        // ...
    }
};
```

Zadanie 7 Zmodyfikuj konstruktory klas `Kolo` i `Kwadrat` tak, aby inicjalizowały odpowiednio obiekt bazowy. Zauważ, że możesz teraz uczynić pole `pole` prywatnym, gdyż odwołujesz się nie bezpośrednio do niego, tylko do konstruktora `Figura`.

Rzutowanie statyczne

Dotychczas mówiliśmy o dziedziczeniu jako sposobie uproszczenia kodu - omijaliśmy wielokrotne definiowanie tych samych metod i pól w klasach pochodnych. Hierarchie dziedziczenia dają nam jednak także możliwość polimorficznego myślenia, tzn. myślenia o obiekcie w kontekście nie tylko jego typu, ale także typów po których dziedziczy (lub które dziedziczą po nim, ale o tym w dalszej części instrukcji).

Zadanie 8 Dodaj do wszystkich 3 stworzonych dotychczas klas metodę `void id()`, która drukuje informację o typie figury oraz jej polu.

Zadanie 9 Napisz wolnostojącą funkcję `void id(const Figura&)`, która przyjmuje obiekt typu `Figura` i woła jego metodę `id`. Stwórz obiekt typu `Kwadrat`. Czy możesz podać go jako argument do funkcji `id`? Jeżeli tak, jaka wiadomość zostanie wydrukowana?

Jak widzimy, wszędzie, gdzie spodziewamy się obiektu klasy bazowej, możemy podać także obiekt klasy pochodnej. Dzieje się tak dzięki niejawnemu rzutowaniu *w górę* hierarchii dziedziczenia (tzn. w stronę klasy bazowej), które wykonuje za nas kompilator. Rzutowanie takie możemy także wykonać jawnie, za pomocą konwersji `static_cast`. Wygląda ona następująco:

```
Pochodna p;
Baza b = static_cast< Baza >( p );
```

Rzutować w ten sposób możemy także wskaźniki oraz referencje.

Zadanie 10 Stwórz obiekt typu `Kwadrat`. Zawołaj jego metodę `id`. Następnie zawołaj metodę `id` jego rzutu na typ `Figura`. Czy wołana jest ta sama metoda?

Dynamiczny polimorfizm

Jak sama nazwa wskazuje, w statycznym polimorfizmie typy, przy pomocy których interpretujemy obiekty, muszą być znane w czasie kompilacji. Powiemy teraz jak decydować o typach obiektów w czasie wykonania programu. Językiem, który do tego wykorzystamy będą wskaźniki i referencje. Jeżeli nie będzie do końca jasne, czemu tak postępujemy, zachęcamy czytelnika do zawieszenia swojego sceptycyzmu do kolejnego rozdziału, w którym pokażemy praktyczne aplikacje przedstawionej tutaj metodologii i wszelkie wątpliwości zostaną rozwiane (a przynajmniej taka jest nadzieja autorów).

Metody wirtualne

Jak widzieliśmy w zadaniach 9. i 10., gdy patrzyliśmy na obiekt przez pryzmat typu, po którym dziedziczy (tzn. uzyskując do niego dostęp przez referencję lub wskaźnik typu bazowego), nie mieliśmy możliwości dostania się do pól i metod tego obiektu pochodzących z jego faktycznej klasy. Jest to zupełnie logiczne - wolnostojącą funkcję `id` moglibyśmy napisać oraz skompilować zanim w ogóle rozpoczęlibyśmy pracę nad `Kwadrat` i `Kolo` (oczywiście zakładając odpowiedni podział na pliki źródłowe i nagłówkowe). `id(const Figura&)` zawoła wtedy metodę `id` klasy `Figura` - żadna inna klasa jeszcze nie istnieje. Naszym celem w tym rozdziale jest przedstawienie sposobu, który pozwoliłby oddalić decyzję o metodzie, która zostanie zawołana do momentu wykonania programu. Możemy wtedy zawołać metodę nie klasy `Figura`, ale klasy obiektu, do którego referencja `const Figura&` tak naprawdę się odnosi. Mechanizmem w C++, który do tego służy, jest słowo kluczowe `virtual`. Klasę, która posiada choć jedną metodę oznaczoną tym słowem nazywamy klasą polimorficzną. W reprezentacji obiektu takiej klasy, kompilator tworzy dodatkowy wskaźnik (*vpointer*), który wskazuje na "prawdziwy" typ danego obiektu (a dokładniej mówiąc, wskazuje miejsce, w którym zapisane jest "prawdziwe" zachowanie obiektu, tzw. *vtable*, o szczegółach można przeczytać np. tutaj).

Zadanie 11 Oznacz metodę `id` klasy `Figura` jako wirtualną. Oznacz metody `id` klas po niej dziedziczących jako nadpisujące (`override`). Powtórz zadanie 9. Czy efekt będzie taki sam jak wcześniej? Czy kod wymagał od Ciebie jakiejkolwiek modyfikacji poza słowami `virtual` i `override`?

Rzutowanie dynamiczne

Ponieważ jesteśmy teraz w stanie dynamicznie określić "prawdziwe" typy obiektów, na które wskazują wskaźniki i do których odnoszą się referencje, to możemy polimorficzne klasy rzutować także w dół oraz na boki hierarchii dziedziczenia (dynamiczne rzutowanie w górę hierarchii jest ekwiwalentne statycznemu). Służy do tego konwersja `dynamic_cast`. Zwróćmy uwagę, że dynamicznie rzutować możemy jedynie wskaźnik i referencje, ale nie obiekty. Jeżeli obiekt, wskaźnik do którego rzutujemy nie jest "tak naprawdę" typu, na który rzutujemy, to wynikiem rzutowania jest wyzerowany wskaźnik (`nullptr`). Przypomnijmy, że wskaźniki są konwertowalne na typ `bool`, także `dynamic_cast` może być np. używany jako mechanizm sprawdzenia, czy wskaźnik klasy bazowej "tak naprawdę" wskazuje na obiekt zadanej klasy pochodnej.

Zadanie 12 Stwórz dynamicznie obiekt typu `Kwadrat` i przypisz go do wskaźnika typu `Figura*` (`Figura* f = new Kwadrat{/* ... */};`). Jaki będzie wynik dynamicznego rzutowania na `f` na `Kwadrat*`, a jaki na `Kolo*`?

Destruktor

W klasach polimorficznych zawsze powinniśmy oznaczać destruktory jako wirtualne. Jeżeli tego nie zrobimy, a obiekt typu pochodnego zostanie usunięty (w znaczeniu `delete`) przy pomocy wskaźnika na typ bazowy, wtedy destruktor typu pochodnego nigdy nie zostanie zawołany. Może to doprowadzić do wycieku zasobów i innych nieprzyjemnych sytuacji.

Zadanie 13 Zdefiniuj dla stworzonych dotychczas klas destruktor, drukujący informację o zniszczeniu obiektu. Zawołaj w funkcji `main`

```
Figura* f = new Kwadrat{/* ... */};
delete f;
```

Które destruktory zostały zawołane? Jak zmieni się sytuacja, gdy uczynisz destruktory wirtualnymi?

Metody i klasy abstrakcyjne

Omówione wyżej metody wirtualne to metody, które możemy nadpisać. Metody abstrakcyjne (zwane też czysto wirtualnymi) to metody, które *musimy* nadpisać. Klasy posiadające takie metody nazywamy abstrakcyjnymi. Nie wolno nam bezpośrednio instancjonować klas abstrakcyjnych, ale możemy oczywiście tworzyć obiekty typów dziedziczących po typach abstrakcyjnych. Metody abstrakcyjne oznaczamy następująco:

```
class C
{
    virtual T metoda(/* argumenty */) = 0;
};
```

Zadanie 14 Stwórz klasę `BytGeometryczny` i zmodyfikuj `Figura` tak, aby po niej dziedziczyła. Dodaj do `BytGeometryczny` abstrakcyjną metodę `void id()`. Co stanie się, gdy spróbujesz stworzyć obiekt typu `BytGeometryczny`? Co stanie się, gdy wykomentujesz z klasy `Figura` metodę `id`?

Metody abstrakcyjne mogą być np. używane przez autorów bibliotek do zdefiniowania schematu, wg. którego ma być używana dana funkcjonalność, a następnie wymuszenia na użytkowniku dostarczenia własnej implementacji tejże funkcjonalności.

Przykłady zastosowań

Przećwiczmy teraz kilka wzorców, które występują często w praktyce programistycznej.

Heterogeniczny kontener

Fakt, że możemy wskazywać na obiekty klasy pochodnej wskaźnikiem do klasy bazowej, może być przez nas wykorzystany do trzymania obiektów różnego typu w jednym miejscu.

Zadanie 15 Napisz klasę `WektorFigur`, która przechowuje tablicę wskaźników do obiektów typu `Figura`. Dla uproszczenia możesz zaalokować pamięć na wskaźniki statycznie (np. 1000 elementów). Dodaj także do klasy licznik, który śledzi, ile obiektów zostało już przypisane.

Zadanie 16 Przeciąż dla klasy `WektorFigur` operator `[]` tak, aby móc indeksować się po trzymanych wskaźnikach. Jeżeli indeks przekroczy liczbę trzymanych obecnie elementów, zwróć `nullptr`.

Zadanie 17 Dodaj do klasy `WektorFigur` destruktor, który zawoła dla każdego trzymanego obiektu operator `delete`.

Zadanie 18 Napisz metodę `push`, która dopisuje na końcu trzymanego zakresu podany wskaźnik i inkrementuje licznik.

Zadanie 19 Napisz metodę `pop`, która niszczy ostatni element trzymanego zakresu i dekrementuje licznik.

W napisanej właśnie klasie `WektorFigur` możemy trzymać różnego typu figury. Zauważmy, że jeżeli dopisalibyśmy nową klasę dziedziczącą po `Figura`, np. trójkąt, nie musimy zmieniać w `WektorFigur` ani jednej linijki. Nasz kod jest zgodny z zasadą *open for extension, closed for modification.*

Fabryka

Fabryka to klasa, której obiekty są używane do tworzenia innych obiektów.

Zadanie 20 Napisz klasę `FabrykaFigur` i przeciąż dla niej operator `()` tak, aby jego sygnaturą było `Figura* operator()(const std::string&, double)`. Jeżeli podanym stringiem jest "kwadrat", niech operator nawiasów konstruuje przy pomocy podanej liczby kwadrat, a jeżeli podano "kolo", niech konstruuje koło. W innym przypadku zwróć `nullptr`.

W taki sposób możemy decydować o typie tworzonych obiektów dopiero w czasie wykonania programu. Nic nie stoi na przeszkodzie, aby argument podawany do fabryki figur był np. wczytywany z klawiatury lub pobierany z sieci.

Wizytator

> Wizytator stanowi nieco bardziej zaawansowany przykład zastosowania polimorfizmu. Poznanie go nie jest niezbędne do kontynuowania bieżącego kursu. Czytelnik powinien w pierwszej kolejności skupić się na solidnym zrozumieniu treści zawartej wyżej. Niemniej jednak, zagadnienie iteracji po elementach różnych typów pojawia się powszechnie i, zdaniem autorów, warto wiedzieć po jakie rozwiązania sięgnąć, gdy taki problem się napotka. Więcej informacji nt. wizytatora dostępne jest np. w artykule na wikipedii, napisanym w dość zrozumiały sposób.

Wizytator to jeden z częściej stosowanych schematów projektowych (ang. *design pattern*) w C++. Mając dany wskaźnik do obiektu polimorficznego, chcielibyśmy potrafić wykonać na nim różne operacje w zależności od typu obiektu, na który "tak naprawdę" wskazuje. Mając dany kontener takich wskaźników, chcielibyśmy móc się po nim odpowiednio przeiterować. Zacznijmy od prostych rzeczy.

Zadanie 21 Dodaj to klasy `WektorFigur` metodę `void idWszystkie()`, która woła po kolei metodę `id` trzymanych figur. Pamiętaj, że mając wskaźnik do obiektu, możesz zawołać jego metodę przy użyciu operatora `->`.

W powyższym zadaniu zakładamy, że chcemy zawołać dla zawartości kontenera konkretą metodę. Jest to dość mało ogólne rozwiązanie. Postawmy się w sytuacji, w której piszemy bibliotekę, z której korzystać będą osoby trzecie. Nie możemy do końca przewidzieć, co będą one chciały zrobić z trzymanymi w kontenerze figurami. Musimy zatem przygotować mechanizm, za pomocą którego użytkownik może zdefiniować własną funkcję, która wykona na obiektach operacje zależne od typu obiektu. Rozwiązaniem tego problemu jest właśnie wizytator.

Zadanie 22 Napisz klasę `WizytatorFigurBaza`. Dodaj do niej 2 abstrakcyjne przeciążenia metody `void wizytuj(...)`: jedno przyjmujące typ `Kwadrat&` i jedno przyjmujące typ `Kolo&`.

Zadanie 23 Dodaj do klasy `Figura` abstrakcyjną metodę `void akceptuj(WizytatorFigurB`

Zadanie 24 Nadpisz w klasach `Kwadrat` i `Kolo` metodę `akceptuj` tak, aby wołała metodę `wizytuj` podanego wizytatora na danym obiekcie geometrycznym (`v.wizytuj(*this);`).

Zadanie 25 Dodaj do klasy `WektorFigur` metodę `void wizytujWszystkie(WizytatorFigu` która woła po kolei dla każdej trzymanej figury metodę `akceptuj` na podanym wizytatorze.

Jako autorzy biblioteki, zakończyliśmy właśnie pracę. Ostatni krok leży po stronie użytkownika.

Zadanie 26 Napisz klasę `WizytatorDrukujacy`, która dziedziczy po klasie `WizytatorFigurBaza`. Nadpisz oba przeciążenia metody `wizytuj` tak, aby drukowały informację o typie wizytowanej figury.

Właśnie w pełni zaimplementowaliśmy wizytator! Aby przetestować jego działanie, wystarczy stworzyć wizytator drukujący oraz kontener figur, wypełnić kontener, a następnie zawołać metodę `wizytujWszystkie` na zdefiniowanym wizytatorze. Zastanówmy się, jak dokładnie działa napisany przez nas kod. Zacznijmy od tego, że podany przez nas wizytator jest z punktu widzenia `wizytujWszystkie` referencją do bazowej klasy wizytatora. Fakt ten pozwala nam ujednolicić interfejs kontenera - jeżeli chcemy zdefiniować nowy wizytator, który robi coś zupełnie innego, wystarczy, że powtórzymy zadanie 26. Nie musimy modyfikować żadnych napisanych wcześniej klas! Ponownie zachowujemy się zgodnie z zasadą *open-closed*. Prześledźmy teraz co dokładnie dzieje się przy wizytowaniu każdej figury. Najpierw wołana jest metoda `akceptuj` figury. Jest ona wirtualna, a zatem - pomimo tego, że nigdzie jawnie nie trzymamy informacji o "prawdziwym" typie figury - wołana jest metoda odpowiedniej klasy pochodnej. Następnie, metoda `akceptuj` klasy pochodnej (`Kwadrat` lub `Kolo`), woła "na sobie" metodę `wizytuj` wizytatora, która przeciążona jest zarówno dla `Kwadrat` jak i dla `Kolo`. Dość charakterystyczna jest tutaj "podwójna delegacja" (ang. *double dispatch*) - wizytator wizytuje, ale także obiekt wizytowany akceptuje. Osiągnęliśmy zatem to, co chcieliśmy - możemy dokonywać na trzymanych figurach dowolnie zdefiniowanych operacji zależnych od "prawdziwego" typu figury.

Zadanie 27 Przetestuj działanie napisanego kodu. Jeżeli masz wątpliwości, że jest ono w pełni dynamiczne możesz uzależnić typ wkładanych do kontenera figur od wejścia z konsoli (pomocna będzie do tego fabryka).