Języki programowania obiektowego

Laboratorium V

Szablony funkcji i klas, zasada odwróconej zależności

szablony funkcji

Szablony funkcji są sposobem na stworzenie funkcji, która może przyjmować argumenty dowolnych typów. Przykładowo można stworzyć funkcję, która mnoży dwie liczby:

```
int pomnoz (int a, int b)
{
	return a * b;
}
```

Funkcja taka działa poprawnie dla argumentów typu `int`, gdy wyniknie potrzeba użycia jej dla typu `float` trzeba będzie ją całą skopiować i zapisać w postaci:

```
float pomnoz (float a, float b)
{
	return a * b;
}
```

Potem skopiujemy ją jeszcze kilkakrotnie dla kolejnych typów które będą nam potrzebne, na przykład `unsigned int`, `char`, `double`. Rozwiązanie to jest mało eleganckie. Ponadto stwarza problemy – jeżeli w pierwszej funkcji wystąpi błąd trzeba będzie go poprawić we wszystkich jej kopiach, lepiej byłoby gdyby można było zrobić to tylko raz. Z pomocą przychodzą szablony funkcji.

```
template <typename T>
T pomnoz (T a, T b)
{
	return a * b;
}
```

Powyższy zapis oznacza, że nie tworzymy funkcji, tylko szablon. Z takiego szablonu będziemy tworzyć funkcje w obrębie programu zastępując słowo `T` pożądanym typem zmiennej. Przyjrzyjmy się temu bliżej. Wyrażenie `template<typename T>` oznacza, że mamy do czynienia z szablonem, który posiada jeden parametr formalny nazwany `T`. Słowo kluczowe `typename` oznacza, że parametr ten jest typem (nazwą typu). Zamiast słowa typename możemy użyć słowa kluczowego `class`. Nazwa tego parametru może być następnie wykorzystywana w definicji funkcji w miejscach, gdzie spodziewamy się nazwy typu. I tak powyższe wyrażenie definiuje funkcję

`pomnoz`, która przyjmuje dwa argumenty typu `T` i zwraca wartość typu `T`, będącą wartością większego z dwu argumentów. Typ `T` jest na razie nie wyspecyfikowany. W tym sensie szablon definiuje całą rodzinę funkcji. Konkretną funkcję z tej rodziny tworzymy poprzez podstawienie za formalny parametr `T` konkretnego typu będącego argumentem szablonu. Takie podstawienie nazywamy konkretyzacją szablonu.

```
template <typename T>
T pomnoz (T a, T b)
{
	return a * b;
}

int main()
{
	float fA = 0.3;
	float fB = 0.4;

	int iA = 2;
	int iB = 4;

	cout << "Wynik mnożenia liczb typu float: " << pomnoz(fA,fB) << endl;
	cout << "Wynik mnożenia liczb typu int: " << pomnoz(iA,iB) << endl;

	return 0;
}
```

Ponieważ w naszym przykładzie wywołaliśmy funkcję `pomnoz` z dwoma argumentami typu float dlatego na podstawie szablonu kompilator stworzy następującą instancję funkcji pomnoz:

```
float pomnoz (float a, float b)
{
	return a * b;
}
```

Szablony mogą posiadać też więcej niż jeden argument:

```
template <typename T1, typename T2>
T2 rzutowanie (T1 A)
{
	return (T2) A;
}

int main()
{
	float fA = 3.3;
	int iA = 0;
	iA = rzutowanie<float, int>(fA);
	cout << iA; // wyświetli 3

}
```

szablony klas

Szablony klas, podobnie jak szablony funkcji służą do zamknięcia wielu bardzo podobnych klas w jednym szablonie. Mamy na przykład zastaw klas przechowujących parę danych:

```
class IntInt
{
public:
```

```
	int a, b;
	int dodaj()
	{
		return a + b;
	}
};

class IntChar
{
public:
	int a;
	char b;
	int dodaj()
	{
		return a + b;
	}
};

class CharChar
{
public:
	char a, b;
	char dodaj()
	{
		return a + b;
	}
};
```

Klasy te są do siebie bardzo podobne, mają identyczne składowe i tą samą metodę `dodaj()`, różnią się tylko typami. Gdybyśmy chcieli stworzyć powyższe klasy dla każdej możliwej pary typów to musielibyśmy stworzyć dużą liczbę podobnych klas co byłoby bardzo niedogodne. Zamiast tworzyć liczne podobne klasy, można stworzyć jeden szablon klasy i na jego podstawie kompilator sam wygeneruje odpowiednie warianty klasy w zależności od tego jakie konkretne typy zostaną użyte w programie. Ogólna postać szablonu klasy wygląda następująco:

```
template<argumenty_szablonowe> class nazwa_klasy
{
  // wnętrze klasy
};
```

Na przykład:

```
template<typename T1, typename T2> class klasa
{
public:
	T1 a;
	T2 b;
	T1 dodaj()
	{
		return a + T1(b);
	}
};
```

W powyższym przykładzie mamy zdefiniowany szablon klasy z dwoma parametrami formalnymi `T1` oraz `T2`. Mając tak zdefiniowany szablon, możemy w oparciu o niego utworzyć obiekt dla dwóch różnych typów:

```
klasa<int, char> obiekt;
```

Stworzyliśmy w ten sposób obiekt typu `klasa` przechowujący elementy typu `int` oraz `char`. W ciele klasy parametr formalny `T1` zostanie zastąpiony typem `int`, zaś `T2` zostanie zastąpione typem `char`.

Jako argumenty szablonowe można także używać zwykłych typów wbudowanych, bądź zdefiniowanych. W takim przypadku argumenty szablonowe stają się stałymi klasy. Na przykład:

```
template<float wspolczynnik> class klasa
{
public:
      int a, b;
      float oblicz()
      {
            return (a + b) * wspolczynnik;
      }
};
```

i obiekt klasy:

```
klasa<0.5> obiekt;
obiekt.a = 2;
obiekt.b = 4;
int wynik = obiekt.oblicz();
```

Powyższa klasa operuje na liczbach naturalnych i posiada funkcję zwracającą liczbę zmiennoprzecinkową. Posiada ona też stały współczynnik typu `float`, w naszym obiekcie równy 0.5.

specjalizacja szablonów klas

Niekiedy zdarza się, iż zdefiniowany wzorzec klasy musi zachowywać się nieco inaczej dla konkretnych typów danych niż w pozostałych przypadkach.

```
template<typename T> class Number
{
private:
  T m_number;
public:
  void print()
  {
    cout << m_number << endl;
  }
};
```

W przypadku gdybyśmy potrzebowali zmodyfikować działanie metody `print()` dla typu `float`, możemy w takim przypadku stworzyć tak zwaną specjalizację szablonu, która będzie implementować odmienne działanie tej klasy dla jakiegoś konkretnego typu. Na przykład:

```
template<typename T> class Number
{
private:
  T m_number;
public:
  void print()
  {
     cout << m_number << endl;
  }
};

template<> class Number<float> // specjalizacja klasy dla typu float
{
private:
  float m_number;
public:
```

```
  void print()
  {
      cout << precision(2) << m_number << endl;
  }
```

Tworząc specjalizację klasy `Number` dla typu `float` informujemy kompilator, że w przypadku gdy utworzymy w programie obiekt klasy `Number<float>` ma on posłużyć się dostarczoną specjalizacją a nie ogólnym szablonem dla pozostałych typów.

Metoda wytwórcza

Metoda wytwórcza (Konstruktor wirtualny, Virtual constructor, Factory Method) jest kreacyjnym wzorcem projektowym, który udostępnia interfejs do tworzenia obiektów w ramach klasy bazowej. Wyobraź sobie, że tworzysz aplikację do zarządzania logistyką. Pierwsza wersja twojej aplikacji pozwala jedynie na obsługę transportu za pośrednictwem ciężarówek, więc większość kodu znajduje się wewnątrz klasy `Ciężarówka`. Po jakimś czasie twoja aplikacja staje się całkiem popularna. Codziennie otrzymujesz tuzin próśb od firm realizujących spedycję morską, abyś dodał stosowną funkcjonalność do swej aplikacji. Świetna wiadomość, prawda? Ale co z kodem? W tej chwili większość twojego kodu jest powiązana z klasą `Ciężarówka`. Dodanie do aplikacji klasy `Statki` wymagałoby dokonania zmian w całym kodzie. Co więcej, jeśli później zdecydujesz się dodać kolejny rodzaj transportu, zapewne będziesz musiał dokonać tych zmian jeszcze jeden raz.

Wzorzec projektowy Metody wytwórczej proponuje zamianę bezpośrednich wywołań konstruktorów obiektów na wywołania specjalnej metody wytwórczej. Jednak nie przejmuj się tym: obiekty nadal powstają, ale teraz dokonuje się to za kulisami — z wnętrza metody wytwórczej. Obiekty zwracane przez metodę wytwórczą często są nazywane *produktami*.

Na pierwszy rzut oka zmiana ta może wydawać się bezcelowa. Przecież przenieśliśmy jedynie wywołanie konstruktora z jednej części programu do drugiej. Ale zwróć uwagę, że teraz możesz nadpisać metodę wytwórczą w podklasie, a tym samym zmienić klasę produktów zwracanych przez metodę. Istnieje jednak małe ograniczenie: podklasy mogą zwracać różne typy produktów tylko wtedy, gdy produkty te mają wspólną klasę bazową lub wspólny interfejs. Ponadto, zwracany typ metody wytwórczej w klasie bazowej powinien być zgodny z tym interfejsem.

Na przykład zarówno klasy `Ciężarówka`, jak i `Statek` powinny implementować interfejs `Transport`, który z kolei deklaruje metodę `dostarczaj()`. Każda klasa różnie implementuje tę metodę: ciężarówki dostarczają towar drogą lądową, statki drogą morską. Metoda wytwórcza znajdująca się w klasie `LogistykaDrogowa` zwraca obiekty `Ciężarówka`, zaś metoda wytwórcza w klasie `LogistykaMorska` zwraca `Statki`.

Kod, który wykorzystuje metodę wytwórczą (zwany często kodem *klienckim*) nie widzi różnicy pomiędzy faktycznymi produktami zwróconymi przez różne podklasy. Klient traktuje wszystkie produkty jako abstrakcyjnie pojęty `Transport`. Klient wie także, że wszystkie obiekty transportowe posiadają metodę `dostarczaj()`, ale szczegóły jej działania nie są dla niego istotne.

```
// Simple factory example

// Pure virtual base class
class Transport
{
public:
    virtual ~Transport() {}

    virtual void deliver() ( void ) = 0;
};

// Derived classes
class Truck : public Transport
```

```
{
public:
    virtual ~Truck()
    {
      cout << "Truck is deleted" << endl;
    }

    // It is also virtual but override is enough to express this (skip virtual)
    void deliver() ( void ) override
    {
      cout << "Truck is delivering ..." << endl;
    }
};

class Ship : public Transport
{
public:
    virtual ~Ship()
    {
      cout << "Ship is deleted" << endl;
    }

    void operator() ( void ) override
    {
      cout << "Ship is delivering ..." << endl;
    }
};


enum EClassId { kTruck, kShip };

auto Factory( EClassId id )
{
switch( id )
{
  case kTruck:
      return unique_ptr<Transport>( make_unique<Truck>() );
  case kShip:
      return unique_ptr<Transport>( make_unique<Ship>() );

      default: assert( false );    // should not be here
    }

    return unique_ptr<Transport>();    // can be empty
}

int main()
{
vector< unique_ptr<Transport> > theObjects;
theObjects.push_back( Factory( kTruck ) );
theObjects.emplace_back( Factory( kShip ) );
// replace Ship with Truck
theObjects[ theObjects.size() - 1 ] = Factory( kTruck );

for( auto & a : theObjects )
   ( * a )deliver(); // call actions via the virtual mechanism
}
```

Zasada odwróconej zależności

Kolejna z zasad SOLID mówi o tym, że klasa wysokiego poziomu nie powinna zależeć od klasy niskiego poziomu. Obydwie powinny zależeć od abstrakcji. Oznacza to, że w deklaracji którejkolwiek klasy, metody czy zmiennej nie powinniśmy używać konkretnych klas a zamiast tego interfejsów lub klas abstrakcyjnych. Rozpatrzmy następujący przykład. Na ogół w bardziej złożonej aplikacji, mamy warstwę programu odpowiedzialną za prezentowanie danych (klasa wysokiego poziomu bo z tą klasą użytkownik ma do czynienia), oraz, aby w ogóle możliwe było przetwarzanie i wyświetlanie danych, warstwę dostępu do tych danych (klasa niskiego poziomu). Zatem ogólnie rzecz ujmując prezentowanie danych jest zależne od dostępu do danych. Zasada odwrócenia zależności odwraca tą logikę – niskopoziomowe klasy (moduły) podłączamy do interfejsów określonych przez moduł wysokiego poziomu (moduł ten decyduje o tym, jakie interfejsy będzie akceptował i tylko obiekty zgodne z tym interfejsem będzie można do niego podpiąć). Ilustruje to poniższy kod.

```
class User
{
};

class Email
{
public:
      void send(const User& user, const string& message)
      {
        //send email notification
      }
};

class NotificationManager
{
public:
      void sendNotification(const User& user, const string& message)
      {
        shared_ptr<Email> email = make_shared<Email>();
        email->send(user,message);
      }
};

int main()
{
      User u;
      NotificationManager manager;
      manager.sendNotification(user, "Hello, world!");

      return 0;
}
```

W powyższym przykładzie, klasa `Email` to moduł niskopoziomowy odpowiadający jedynie za wysyłanie powiadomień. Natomiast klasa `NotificationManager` to moduł wysokiego poziomu, ponieważ to z tym modułem użytkownik będzie mieć bezpośrednio do czynienia. Zobaczmy, że w powyższym przykładzie wiążemy metodę wysyłania powiadomień (funkcja `send()` w klasie `Email`) z menedżerem powiadomień przez tworzenie obiektu `email` w klasie `NotificationManager`. W ten sposób klasa wysokiego poziomu (`NotificationManager`) zależy bezpośrednio od klasy niskiego poziomu (`Email`). Technicznie jest to poprawne, ponieważ kod oczywiście zadziała, jednak problemem jest właśnie to powiązanie gdyż tworzy ono silne sprzężenie pomiędzy klasami niskiego i wysokiego poziomu. Zobaczmy zatem, jak możemy zmodyfikować powyższy kod tak aby stosował on zasadę odwróconej zależności, usuwając tym samym silne sprzężenie pomiędzy klasami wysokiego i niskiego poziomu.

```
class User
{
};

class INotificationService  // wprowadzamy odpowiedni interfejs
{
public:
      virtual void send(const User& user, const string& message) = 0;
};

class Email : public INotificationService // klasa Email implementuje ten interfejs
{
public:
      void send(const User& user, const string& message) override
      {
        // send email notification
      }
};

class NotificationManager
{
public:
      void sendNotification(const User& user, const INotificationService&
notificationService, const string& message)
      {
        notificationService.send(user, message);
      }
};
```

Aby dostosować nasz przykład do wymagań zasady odwróconej zależności, wprowadziliśmy interfejs `INotificationService`, który zawiera metodę czysto wirtualną `send()`. Następnie klasa `Email` dziedziczy po tym interfejsie i implementuje metodę `send()`, która przesyła powiadomienie za pomocą emaila. Na koniec, klasa wysokiego poziomu `NotificationManager` nie zależy już bezpośrednio od klasy `Email` ale zamiast tego w metodzie `SendNotification` pobiera dodatkowy wskaźnik na obiekt dziedziczący z interfejsu `INotificationService`. Dzięki powyższym zamianom odwróciliśmy wcześniejsze zależności i zyskaliśmy większą elastyczność. Najważniejszym skutkiem wprowadzonych zmian jest to, że jeżeli będziemy chcieli dodać inny rodzaj wysyłania powiadomień, to wystarczy dodać odpowiednią klasę (która implementuje stworzony interfejs `INotificationService`) bez konieczności ingerencji w już istniejącą klasę `NotificationManager`.

zadania

Write the code for each of the tasks below in one source file.

1. Create a template function named `solve_equation` which prints the solution of the quadratic equation $ax^2+bx+c=0$. In the `main` function do the following:
   a. Create three variables `a` ,`b`, `c` of type `double`,
   b. ask the user to input from the keyboard values for these variables and then print the equation solution for the parameters `a`, `b`, `c`.

c. change the variables type (e. g. from `double` to `float` or `int`) and call the `solve_equation` function again.

2. Create a template class named `ValueContainer` which stores one value of any type. The class should have:
   a. parameterized constructor (with default value),
   b. an overloaded prefix increment operator `++` which returns the stored value increased by one.
   c. overloaded output operator.

   In the `main` function create `ValueContainer` object initialized with an integer value. and print on the screen the incremented value. Next, initialize the object with a character '`b`' and print on the screen the incremented value.

3. Create a template specialization of the class `ValueContainer` for type `char`. In this case the overloaded increment operator should return the stored character in the form of the upper case. This can be achieved by using the `std::toupper()` function. In the `main` function, create two `ValueContainer` objects. The first is initialized with any integer value and the second one is initialized with a character '`j`'. Print the incremented values on the screen.

4. Create a template function named `PowIt` with a default template type of `float` and one non-type parameter named `p` which defaults to 3. The function should take one argument named `x` and return the value of x^p. In the main function call the templated function with and without default types and arguments.

5. Write a template class `ArrayUtils` with the following template methods
   a. `print(T[], size)` - should be able accept any array and print it
   b. `printReverse (T[], size)` - should be able to accept any array and print it in reverse
   c. `sumAll(T[], size)` - sum all the elements in the array and return the total
   d. `productOfArray(T[], size)` - return the product of all the elements in the array
   e. Test your template class with different types of arrays such as arrays of ints, floats, strings etc.

1. Wykorzystując wzorzec projektowy metoda wytwórcza, zaimplementuj poniższy system.