

Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 5

31 października 2019

- przykład
- przypomnienie
- organizacja kodu w plikach
- szablon funkcji
- szablon klasy
- szablon metody
- specjalizacja szablonu

```
class Polygon
{
public:
    Polygon(size_t size);
    Polygon(const Polygon&);
    Polygon(Polygon&&);
    Polygon& operator=(const Polygon&);
    Polygon& operator=(Polygon&&);
    void push(Point);
    size_t numberOfPoints() const;
    Point& operator[](size_t index);
    Point operator[](size_t index) const;
    ~Polygon();
private:
    Polygon() = default;
    Polygon(size_t size, size_t number_of_points);
    void swap(Polygon&, Polygon&);
    size_t size_ = 0;
    size_t number_of_points_ = 0;
    Point* points_ = nullptr;
};
```

```
#include <random>
#include "polygon.hpp"

auto get_point()
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(-10.0, 10.0);

    return Point{ dis(gen), dis(gen) };
}

int main(void)
{
    auto polygon = Polygon(2);
    polygon.push(get_point()); // ok
    polygon.push(get_point()); // ok
    polygon.push(get_point()); // blad

    return 0;
}
```

Problem

Należy przygotować moduł reprezentujący wielokąt.

Implementowany moduł ma umożliwiać:

- dodanie dowolnej ilości punktów,
- dostęp do dowolnego dodanego już punktu w stałym czasie,
- uzyskanie informacji na temat ilości dostanych już punktów.

Taki wielokąt implementowany jest przez wiele edytorów graficznych umożliwiających tworzenie wielokątów poprzez klikanie w różne punkty obszaru rysowania.

Dodatkowo należy uwzględnić:

- prawidłowe zarządzanie pamięcią,
- prawidłowe operacje kopiowania oraz przenoszenia.

```
#include "polygon.hpp"

class ExtendablePolygon
{
public:
    ExtendablePolygon() = default;

    void push(const Point&);
    size_t numberOfPoints() const;
    Point& operator[](size_t index);
    Point operator[](size_t index) const;

private:
    ExtendablePolygon(const Polygon&, size_t memory_size);

    void _extendMemorySize();

    Polygon polygon_ = Polygon(0);
    static const size_t increment_memory_size_ = 2;
};
```

```
size_t ExtendablePolygon::numberOfPoints() const
{
    return polygon_.numberOfPoints();
}

Point & ExtendablePolygon::operator[](size_t index)
{
    return polygon_[index];
}

Point ExtendablePolygon::operator[](size_t index) const
{
    return polygon_[index];
}
```

```
void ExtendablePolygon::push(const Point& point)
{
    if (polygon_.isFull())
    {
        _extendMemorySize();
    }

    polygon_.push(point);
}
```

polygon.hpp

```
class Polygon
{
    /* ... */
public:
    bool isFull();
    /* ... */
};
```

polygon.cpp

```
bool Polygon::isFull()
{
    return number_of_points_ == size_;
}
```



```
void ExtendablePolygon::_extendMemorySize()
{
    *this = ExtendablePolygon(polygon_,
        polygon_.numberOfPoints() + increment_memory_size_);
}

ExtendablePolygon::ExtendablePolygon(const Polygon & polygon,
    size_t polygon_size)
    : polygon_(polygon_size)
{
    for (size_t i = 0; i < polygon.numberOfPoints(); ++i)
    {
        polygon_.push(polygon[i]);
    }
}
```

W jaki sposób można ograniczyć liczbę kopiowań? Jaki jest koszt takiego rozwiązania?

Istotę programowania obiektowego stanowią obiekty i ich wzajemne relacje w trakcie działania programu.

Klasa stanowi opis tego jak obiekty są tworzone i niszczone, jakie mogą mieć stany wewnętrzne oraz w jaki sposób współpracują z innymi obiektami obecnymi w programie.

Klasa

```
class Point {
    double x, y;
    /*...*/
public:
    Point (double x, double y);
    /*...*/
};
```

Obiekt

```
int main(void)
{
    Point p1(1, 1);
    Point p2 = p1 + p1;
    /*...*/
    return 0;
}
```

W języku C++ istnieje podział na trzy główne typy obiektów:

- typ zwykły,
- typ referencyjny,
- typ wskaźnikowy.

Typ referencyjny może być tak zwaną l-referencją lub r-referencją.

```
int main(void)
{
    Point p(1, 1);           // obiekt
    Point* wp = &p;         // wskaźnik
    Point& rp = p;          // l-referencja
    std::move(p);           // r-referencja
    return 0;
}
```

W języku C++ istnieje możliwość definiowania wolnych funkcji, tj. funkcji niezwiązanych z żadnym obiektem bądź klasą. Funkcje te mogą przyjmować postać zwykłą bądź postać operatora. Funkcje mogą przyjmować dowolną ilość argumentów dowolnego typu oraz zwracać tylko jedną wartość. Mogą także mieć takie same nazwy (o ile różnią się typem argumentów) - polimorfizm statyczny.

```
void print_info(const char* str)
{
    std::cout << str << std::endl;
}
```

```
Point operator+(const Point& lhs, const Point& rhs)
{
    return Point(lhs.x + rhs.x, lhs.y + rhs.y);
}
```

Wszystkie metody niestaticzne są to **funkcje**, wywoływane ma rzecz danego obiektu, w których obiekt ten jest niejawnym argumentem danym przez wskaźnik **this**.

Metody mogą mieć modyfikator **const**, który informuje kompilator (użytkownika), że dana metoda nie zmienia **stanu wewnętrznego** obiektu.

Metody statyczne mogą modyfikować jedynie statyczne pola klasy, nie są związane z żadnym konkretnym obiektem danego typu.

```
class TrainBuilder
{
    Locomotive buildLocomotive(/*...*/) const;
    TrainWagon buildTrainWagon(/*...*/) const;
public:
    TrainBuilder(/*...*/);
    Train build(const TrainSpec& spec) const;
};
```

Wszystkie pola (inaczej atrybuty) klasy pomijając pola statyczne (`static`) oraz modyfikowalne (`mutable`) opisują stan wewnętrzny obiektu.

Podobnie jak metody, pola również mogą mieć atrybut stałości (`const`)

```
class MemoryChunk
{
private:
    struct MemorySize { size_t size; };

    const double * memory_;
    const MemorySize size_;

public:
    MemoryChunk(const double* mem, const size_t size);
    const double* data(void) const;
    const double& operator [] (size_t index) const;
};
```

Zazwyczaj klasy implementowane są w dwóch rodzajach plików

- pliki nagłówkowe (ang. *header*) z rozszerzeniem `.hpp` lub `.h`
- pliki źródłowe (ang. *source*) z rozszerzeniem `.cpp`, `.c` lub `.cc`

Plik nagłówkowy stanowi interfejs danego modułu (moduł może zawierać więcej niż jedną klasę), dołączany (dyrektywa `#include`) do innych modułów. Wszystko co zawiera plik nagłówkowy jest widoczne dla innych użytkowników i jest niejako publiczne (nawet jeśli zadeklarowane w obszarze prywatnym - `private`).

Plik źródłowy może być kompilowany do pliku obiektowego (zawierającego kod binarny) a następnie dołączany do innych jednostek translacji w trakcie konsolidacji w celu utworzenia pliku wykonywalnego. Pliki obiektowe, biblioteki statyczne i dynamiczne nie są czytelne dla ich użytkowników - zawierają kod maszynowy.

- jeden z paradygmatów programowania
- możliwe jest tworzenie kodu (algorytmu) bez znajomości typów danych na jakich będzie operował - o ile algorytm na to pozwala.
- możliwe jest opracowanie wzorców typów danych, które będą się zachowywać w różny sposób w zależności od lokalnej potrzeby użytkownika.

Paradygmat programowania to inaczej sposób w jaki programista patrzy na program i w jak steruje jego przepływami sterowania. W programowaniu obiektowym program traktowany jest jako współpracujące ze sobą obiekty.

Jakie są przykłady algorytmów, które są poprawne niezależnie od typu danych?

Potęgowanie za pomocą mnożenia

$$x^k = x \cdot x^{k-1} = \underbrace{x \cdot x \cdot x \cdot \dots \cdot x}_k \quad (1)$$

Algorytm szybkiego potęgowania

$$x^n = \begin{cases} x \cdot (x^2)^{\frac{n-1}{2}}, & \text{jeśli } n \text{ jest nieparzyste} \\ (x^2)^{\frac{n}{2}}, & \text{jeśli } n \text{ jest parzyste} \end{cases} \quad (2)$$

problem obliczania potęgi o wykładniku naturalnym

```
double power(double x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}

int main(void)
{
    for (unsigned int i = 0; i < 10; ++i)
        std::cout << power(2, i) << std::endl;
    return 0;
}
```

Co musimy zrobić gdy chcemy podnosić do potęgi tylko liczby całkowite, reprezentowane na przykład przez typ `long long` ?
Dlaczego chcielibyśmy móc to robić?

problem obliczania potęgi o wykładniku naturalnym

```
long long power(long long x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}
```

problem obliczania potęgi o wykładniku naturalnym

```
long long power(long long x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}
```

```
template<typename T>
T power(T x, unsigned int n)
{
    if (0 == n) return 1;
    if (1 == n) return x;

    if (n & 1) return x * power(x * x, (n - 1) / 2);
    else return power(x * x, n / 2);
}
```

Szablony (ang. *template*) w języku C++ oznaczane są przy pomocy słowa kluczowego `template`. Deklaracja lub definicja funkcji (lub obiektu, metody) szablonej poprzedza konstrukcja:

```
template<typename T>
```

lub:

```
template<class T>
```

W trakcie kompilacji, kod odpowiedniej funkcji zostaje wygenerowany na podstawie przekazanych argumentów szablonu. Każda funkcja wygenerowana w ten sposób posiada osobny kod binarny w programie i osobny adres w pamięci.

- Argumenty szablonu mogą być podawane w sposób jawny za nazwą wołanej funkcji, argument ten określa typ dla którego konkretyzowany jest szablon:

```
auto r1 = power<int>(2, 4);
```

- Jeśli kompilator na podstawie wywołania jest w stanie wydedukować typy argumentów szablonu, wtedy argumenty te mogą być pominięte w wywołaniu; Automatyczna dedukcja typów argumentów szablonu wyłącza automatyczne konwersje:

```
auto r2 = power(2, 4);  
auto r3 = power(2.1, 4);
```

- Podanie argumentów w sposób jawny, z kolei umożliwia automatyczną konwersję typów:

```
auto r2 = power<double>(2, 4);  
auto r3 = power<int>(2.1, 4);
```

- Automatyczna dedukcja parametrów szablonu jest możliwa jedynie wtedy, gdy parametry wywołania funkcji zależą od typów parametrów szablonu.
- Jeżeli parametr szablonu określa na przykład typ zwracany przez funkcję musi być on podany jawnie wraz z wywołaniem funkcji.
- Liczba parametrów szablonu nie jest w żaden sposób ograniczona.
- Próba konkretyzacji poprawnego szablonu w nieprawidłowymi typami argumentów, może generować błąd kompilacji

```
//auto r4 = power(" napis", 4); // blad kompilacji
```

Dlaczego powyższa instrukcja jest nieprawidłowa a poniższa już tak:

```
auto r4 = power('n', 4);
```

W celu wywołania odpowiedniej funkcji kompilator postępuje według następujących kroków:

- 1 Poszukiwana jest funkcja o argumentach dokładnie pasujących do tych w wywołaniu
- 2 Poszukiwany jest wzorzec funkcji pozwalający na generację funkcji spełniającej warunek pierwszy
- 3 Poszukiwana jest funkcja przeciążona o pasujących argumentach
- 4 Zgłaszany jest błąd.

Podobnie jak w przypadku funkcji istnieje możliwość definiowania wzorców klas.

```
template<typename T>
class MemoryT
{
public:
    struct Size { size_t size; };

    MemoryT(Size size);

    T& operator[](size_t i);
    const T& operator[](size_t i) const;

    ~MemoryT(void);
private:
    T* data_;
    Size size_;

    MemoryT(const MemoryT&) = delete;
    MemoryT(MemoryT&&) = delete;
};
```

W przypadku szablonu klasy nie istnieje możliwość automatycznej dedukcji argumentów szablonu. Należy podać je w sposób jawny.

```
int main(void){
    size_t N = 10;
    MemoryT<double> mem1({ N });
    MemoryT<char> mem2({ N });
    // MemoryT mem3({ N }); // blad
};
```

Istnieje możliwość podania argumentów domyślnych

```
template<typename T = double>
class MemoryT { /* ... */ };

int main(void){
    size_t N = 10;
    MemoryT mem3({ N }); // OK
};
```

Parametrami szablonów mogą być również stałe niebędące typami (np. liczby całkowite, typ wyliczeniowy)

```
template<typename T, size_t N = 10>
class MemoryT2
{
public:
    T& operator[](size_t i) ;
    const T& operator[](size_t i) const;

private:
    T data_[N];
};
```

Uwaga. Każda nowa specjalizacja szablonu powoduje powstanie nowego typu danych. Typy te, o ile nie są zaprzyjaźnione, nic nie wiedzą o pozostałych specjalizacjach z danej rodziny. W szczególności nie istnieje możliwość kopiowania danych pomiędzy obiektami różnych typów.

```
MemoryT2<double, 10> mem4;  
mem4[0] = 0; // ...  
MemoryT2<double, 10> mem5 = mem4; // OK  
// MemoryT2<double, 11> mem6 = mem4; // blad  
// MemoryT2<float, 10> mem7 = mem4; // blad
```

definicja metod szablonu klas poza ciałem szablonu

```
template<typename T>
class MemoryT
{
public:
    /* ... */
    MemoryT( const MemoryT&);
    MemoryT( MemoryT&&);
    /* ... */
};

template<typename T>
MemoryT<T>::MemoryT<T>(const MemoryT<T>& rhs)
    : MemoryT<T>(rhs.size_)
{
    memcpy(data_, rhs.data_, rhs.size_.size * sizeof(T));
}

template<typename T>
MemoryT<T>::MemoryT<T>(MemoryT<T>&& rhs)
    : data_(rhs.data_), size_(std::move(rhs.size_))
{
    rhs.data_ = nullptr;
    rhs.size_.size = 0;
}
```

W języku C++ istnieje również możliwość definiowania metod szablonowych zarówno w klasach jak i wzorcach klas.

```
template<typename T>
class MemoryT {
public:
    template<typename U>
    MemoryT(const MemoryT<U>& rhs);

    friend class MemoryT;
    /* ... */
};

template<typename T>
template<typename U>
MemoryT<T>::MemoryT(const MemoryT<U>& rhs)
    : MemoryT<T>({rhs.size_, size})
{
    for (size_t i = 0; i < size_.size; ++i)
        data_[i] = static_cast<T>(rhs.data_[i]);
}
```

W poniższym przypadku zdefiniowany został konstruktor pozwalający na konwersję jednego typu w drugi. Taki konstruktor pomimo, że przypomina konstruktor kopiujący jest zwykłym konstruktorem parametrycznym. Dlaczego?

```
int main( void )
{
    MemoryT<double> mem_d({ N });
    for ( int i = 0; i < N; ++i )
        mem_d[ i ] = i + 0.1;

    MemoryT<int> mem_i(mem_d);

    for ( int i = 0; i < N; ++i )
        std::cout << mem_d[ i ] << "\n" << mem_i[ i ] << std::endl;

    // MemoryT<std::string> mem_s = mem_d; // blad

    return 0;
}
```

Konstruktor taki jak w przykładzie pozwala na automatyczną konwersję pomiędzy różnymi typami. Jego implementacja może być kontrowersyjna, gdyż kompilator ma możliwość zastosowania automatycznej konwersji. Pozostawienie takiego konstruktora pozostaje w decyzji programisty / projektanta systemu.

```
template<typename T>
class MemoryT
{
private:
    template<typename U>
    MemoryT(const MemoryT<U>& rhs);
    friend class MemoryT;
public:
    template<typename U>
    MemoryT<U> to(void);

    /*...*/
};
```

Konstruktor pozwalającego na utworzenie obszaru pamięci na podstawie obiektu będącego instancją innej specjalizacji klasy szablonowej został zadeklarowany w zakresie prywatnym w związku z tym kompilator nie będzie mógł użyć niejawniej konwersji - metoda `::to` będzie musiała być wywołana w sposób jawny.


```
template<typename T>
template<typename U>
MemoryT<U> MemoryT<T>::to(void)
{
    return MemoryT<U>(*this);
}
```

Pozostawienie tego konstruktora, umożliwia zaimplementowanie w prosty sposób metody konwertującej `::to`. Niezbędna w tym przypadku jest deklaracja przyjaźni z pozostałymi specjalizacjami klasy szablonowej.

```
int main(void)
{
    MemoryT<double> mem_d({ N });
    for (int i = 0; i < N; ++i) mem_d[i] = i+0.1;

    //MemoryT<int> mem_i(mem_d);
    auto mem_i = mem_d.to<int>();
    return 0;
}
```

memory_t.hpp

```
#ifndef MEMORY_T_HPP
#define MEMORY_T_HPP
template<typename T>
class MemoryT {
    /*...*/
public:
    template<typename U>
    MemoryT<U> to(void);
    /*...*/
};

#include "memory_t.hpp"
#endif
```

memory_t.hpp

```
#ifdef MEMORY_T_HPP
/*...*/
template<typename T>
template<typename U>
MemoryT<T>::MemoryT<T>(const MemoryT<U>& rhs)
    : MemoryT<T>({ rhs.size_, size }) { /* ... */ }
/*...*/
#endif
```

```
int main( void )
{
    MemoryT<double> mem_d( { N } );
    for ( int i = 0; i < N; ++i )
        mem_d[ i ] = i + 0.1;

    auto mem_s = mem_d.to<std::string>();

    return 0;
}
```

Jak sprawić by konwersja na napis była możliwa?

Język C++ umożliwia specjalizację szablonów częściową bądź pełną.

```
template<typename T> template<typename U>
MemoryT<U> MemoryT<T>::to(void) { return MemoryT<U>(*this); }

template<> template<typename U>
MemoryT<std::string>::MemoryT<std::string>(const MemoryT<U>& rhs)
: MemoryT<std::string>({ rhs.size_ .size }) {
    for (size_t i = 0; i < size_ .size; ++i)
        data_[i] = std::to_string(rhs.data_[i]);
}
```

Tego typu konstrukcje pozwalają na wyspecyfikowanie w jaki sposób ma zachowywać się klasa szablonowa dla niektórych konkretnych typów parametrów.

Jeżeli wymagana jest zbyt duża ilość specjalizacji być może zastosowanie klasy szablonowej nie jest właściwym sposobem rozwiązania problemu.