Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 7

14 listopada 2019

Treść dzisiejszego wykładu

- przykład
- funkcje i funktory
- wstęp do wyrażeń lambda
- `std::function`
- algorytmy z biblioteki standardowej
- sprytne wskaźniki

przykład

Zadanie polega na przygotowaniu klasy reprezentującej rozgrywki ligowe (np. sportowych gier zespołowych) która udostępnia następującą funkcjonalność:

- dodanie drużyny do listy zespołów biorących udział z rozgrywkach
- dodanie wyniku rozegranego meczu
- pobranie i wyświetlenie rankingu
- pobranie i wyświetlenie spotkań rozegranych oraz pozostałych do rozegrania

```
template<typename T>
bool almost_equal_to_fun(const T& lhs, const T& rhs, int ulp = 2)
{
    static_assert(std::is_floating_point_v<T>,
        "T must be floating point type");

    auto limit =  std::numeric_limits<T>::epsilon();
    auto min = std::numeric_limits<T>::min();

    return std::abs(lhs - rhs) <= limit * std::abs(lhs + rhs) * ul
        || std::abs(lhs - rhs) < min;
}
```

funktor

```
struct almost_equal_to
{
    template<typename T>
    bool operator()(const T& lhs, const T& rhs, int ulp = 2)
    {
        calls++;
        static_assert(std::is_floating_point_v<T>,
            "T must be floating point type");

        auto limit =  std::numeric_limits<T>::epsilon();
        auto min = std::numeric_limits<T>::min();

        return std::abs(lhs - rhs) <= limit * std::abs(lhs + rhs)
            || std::abs(lhs - rhs) < min;
    }

    unsigned int calls = 0;
};
```

funktor

Funktor jest to obiekt, który może być wywołany jak zwykła funkcja. Każdy obiekt klasy, która posiada zdefiniowany operator wywołania (`R K::operator ()(S a, T b, ...);`) może pełnić rolę funktora (obiektu funkcyjnego). Zaletą, którą posiadają funktory, a której nie posiadają zwykłe funkcje jest możliwość posiadania stanu wewnętrznego.

```
int main()
{
    double a = 0.2;
    double b = 1 / std::sqrt(5.0) / std::sqrt(5.0);

    auto equal_compare = almost_equal_to();

    std::cout << a == b << '\n';                // false
    std::cout << equal_compare(a,b) << '\n'; // true
}
```

funktor

Funktor jest to obiekt, który może być wywołany jak zwykła funkcja. Każdy obiekt klasy, która posiada zdefiniowany operator wywołania (`R K::operator ()(S a, T b, ...);`) może pełnić rolę funktora (obiektu funkcyjnego). Zaletą, którą posiadają funktory, a której nie posiadają zwykłe funkcje jest możliwość posiadania stanu wewnętrznego.

```
int main()
{
    int a = 1;
    int b = 5 / std::sqrt(5.0) / std::sqrt(5.0);

    auto equal_compare = almost_equal_to();

    std::cout << a == b << '\n';              // true
//  std::cout << equal_compare(a,b) << '\n'; // blad kompilacji
}
```

```
struct equal_to
{
    template<typename T>
    bool operator()(const T& lhs, const T& rhs) const
    {
        if constexpr (std::is_floating_point_v<T>)
        {
            return almost_equal_to()(lhs, rhs, 2);
        }
        else
        {
            return lhs == rhs;
        }
    }
};
```

Wyrażenie `std::is_floating_point_v<T>` sprawdza (w trakcie kompilacji) czy typ argumentów przekazanych do metody klasy `equal_to` jest liczbą zmiennoprzecinkową (`float`, `double`, ...).

```
int main()
{
    double a = 0.2;
    double b = 1 / std::sqrt(5.0) / std::sqrt(5.0);
    int c = 1;
    int d = 1;

    auto equal_compare = equal_to();

    std::cout << a == b << '\n';                  // false
    std::cout << equal_compare(a,b) << '\n'; // true
    std::cout << c == d << '\n';                  // true
    std::cout << equal_compare(c,d) << '\n'; // true
}
```

W jaki sposób należałoby zaimplementować pozostałe operacje porównania (greater_than, ...)

wyrażenia lambda

Wyrażenia lambda pozwala zdefiniować anonimową funkcję w miejscu jej użycia. Wyrażenia te posiadają swój typ - są to klasy. Jednak klasy te nie posiadają nazw (posiadają nazwy unikalne znane jedynie w trakcie kompilacji). Obiekty reprezentujące wyrażenia lambda mają semantykę prawych wartości (nie można im przypisać innych wartości).
Wyrażenia lambda pozwalają uniknąć tworzenia funkcji, klas lub metod jedynie w celu pojedynczego ich wykorzystania. Przykładem, w którym mogą być zastosowane wyrażenia lambda jest sortowanie elementów w kolekcji.

```
template<typename Container, class Compare>
Container sort(const Container& arg, Compare cmp)
{
    /* implementacja sortowania */
}
```

wyrażenia lambda

Schemat wyrażenia lambda:

```
[ elementy_przechwytywane] (parametry) -> typ_zwracany
{
    cialo_wyrazenia
}
```

- elementy przechwytywane (opcjonalne), rozdzielone przecinkami wartości znane poza wyrażeniem lambda przekazywane przez wartość lub referencję; również `this`.
- parametry, argumenty funkcji (dowolna ilość).
- typ zwracany (opcjonalnie) - typ obiektu zwracanego przez wyrażenie lambda.
- ciało wyrażenia - implementacja wyrażenia lambda.

https://en.cppreference.com/w/cpp/language/lambda

wyrażenia lambda

```
#include <vector>
#include <algorithm>

template<typename Container, class Compare>
Container sort(const Container& arg, Compare cmp)
{
    auto result = arg;
    std::sort(result.begin(), result.end(), cmp);
    return result;
}

int main(void)
{
    std::vector<int> v0{ 4,3,2,5,7,1 };

    auto v1 = sort(v0, [](int a, int b)->bool { return a < b; });
    // 1 2 3 4 5 7
    auto v2 = sort(v0, [](int a, int b)->bool { return a > b; });
    // 7 5 4 3 2 1

    return 0;
}
```

std::function

Klasa szablonowa `std::function` jest polimorficznym wraperem ogólnego przeznaczenia. Poszczególne Instancje `std::function` potrafią przechowywać, kopiować oraz wywoływać dowolny byt, który może być wywoływany: funkcja, wyrażenie lambda, obiekt funkcyjny jak również wskaźnik do składowej klasy.

```
template< class R, class... Args >
class function<R(Args...)>;
```

- `R` - typ zwracany przez wyrażenie
- `Args` - argumenty przekazywane do wyrażenia

https://en.cppreference.com/w/cpp/utility/functional/function

std::function

```
template<typename T>
using Compare = std::function<bool(const T&, const T&)>;

int main(void)
{
    std::vector<Compare<double>> comparators;

    comparators.push_back(equal_to_fun<double>);
    comparators.push_back(equal_to());
    comparators.push_back( [](const double& a, const double& b)
        ->bool {return a < b; });

    std::vector<double> v0
        { 0.1, 0.2, 1.0/std::sqrt(5.0)/std::sqrt(5.0), 0.3 };

    for (auto & comparator : comparators)
    {
        v0 = sort(v0, comparator);
    }

    return 0;
}
```

biblioteka algorithm

- algorytmy ogólnego przeznaczenia wykorzystywane do operacji wykonywanych na kontenerach z biblioteki standardowej
- wykorzystywane są iteratory do określania zakresy na jaki wykonywana jest operacja
- poszczególne algorytmy:
  - mogą być wykonywane w miejscu (np. `std::sort`, `std::unique`, `std::reverse`)
  - mogą zwracać iterator wskazujący na konkretny element w kolekcji (np. `std::find`, `std::find_if`, `std::min_element`)
  - mogą zwracać wartość (np. `std::accumulate`)
- dobrą praktyką jest przed przystąpieniem do implementacji jakiejś funkcjonalności jest sprawdzenie czy któregoś z algorytmów z biblioteki standardowej nie da się do tego zaadaptować

https://en.cppreference.com/w/cpp/algorithm

przykładowe algorytmy

- sprawdzające zbiór, niemodyfikujące (np. `all_of`, `for_each`, `find`, `count`)
- modyfikujące uporządkowanie zbioru (np. `sort`, `unique`, `replace`, `reverse`)
- operacje na posortowanych zbiorach(np. `merge`, `includes`, `set_difference`)
- znajdowanie elementów największych (najmniejszych)(np. `max`, `min_element`)
- permutacje zbiorów(np. `next_permutation`)
- operacje numeryczne (np. `inner_product`, `reduce`)
- ...

przykład użycia

```
#include <iostream>
#include <vector>
#include <algorithm>

template<typename T>
static bool compare(T& a, T& b)
{
    return a < b;
}

int main(void)
{
    std::vector<int> v1 = {1, 3, 4, -7, 6, 12, -13};

    for(auto &v : v1)
        std::cout << v << ' ';
    std::cout << std::endl;

    std::cout << *std::max_element(v1.begin(), v1.end(), compare<int> );
    std::cout << std::endl;

    return 0;
}
```

```
template<typename T>
static bool compare(T& a, T& b)
{
    return a < b;
}

*std::max_element(v1.begin(), v1.end(), compare<int> );
```

```
template<typename T>
static bool compare(T& a, T& b)
{
    return a < b;
}

*std::max_element(v1.begin(), v1.end(), compare<int> );
```

```
*std::max_element(v1.begin(), v1.end(),
      [](const int & a, const int & b )->bool { return a < b; } );
```

```
template<typename T>
static bool compare(T& a, T& b)
{
    return a < b;
}

*std::max_element(v1.begin(), v1.end(), compare<int> );
```

```
*std::max_element(v1.begin(), v1.end(),
      [](const int & a, const int & b )->bool { return a < b; } );
```

```
*std::min_element(v1.begin(), v1.end(),
      [](const int & a, const int & b )->bool { return a < b; } );

*std::max_element(v1.begin(), v1.end(),
      [](const int & a, const int & b )->bool { return a > b; } );
```

```
template<typename T>
static bool compare(T& a, T& b)
{
    return a < b;
}

*std::max_element(v1.begin(), v1.end(), compare<int> );
```

```
*std::max_element(v1.begin(), v1.end(),
      []( const int & a, const int & b )->bool { return a < b; } );
```

```
*std::min_element(v1.begin(), v1.end(),
      []( const int & a, const int & b )->bool { return a < b; } );

*std::max_element(v1.begin(), v1.end(),
      []( const int & a, const int & b )->bool { return a > b; } );
```

```
std::for_each(v1.begin(), v1.end(), [](int& i) { i ++;});
```

```
std::sort( v1.begin(), v1.end(),
    [](  const int & a, const int & b )->bool { return a < b; } );

std::sort( v1.begin(), v1.end(),
    [](  const int & a, const int & b )->bool { return a > b; } );
```

```
std::sort( v1.begin(), v1.end(),
    [](  const int & a, const int & b )->bool { return a < b; } );

std::sort( v1.begin(), v1.end(),
    [](  const int & a, const int & b )->bool { return a > b; } );
```

```
std::find_if(v1.begin(), v1.end(),
    [](const int &a) -> bool {return a == 4;}) - v1.begin();

std::find_if(v1.begin()+2, v1.end()-2,
    [](const int &a) -> bool {return a == 4;}) - v1.begin();
```

```
std::sort( v1.begin(), v1.end(),
    [](  const int & a, const int & b )->bool { return a < b; } );

std::sort( v1.begin(), v1.end(),
    [](  const int & a, const int & b )->bool { return a > b; } );
```

```
std::find_if(v1.begin(), v1.end(),
    [](const int &a) -> bool {return a == 4;}) - v1.begin();

std::find_if(v1.begin()+2, v1.end()-2,
    [](const int &a) -> bool {return a == 4;}) - v1.begin();
```

```
std::accumulate(v1.begin(), v1.end(),0);

std::accumulate(v1.begin(), v1.end(),1,
    [](const int &a, const int& b) -> int { return a*b;});
```

wskaźniki

+ efektywne
+ wygodne, zwłaszcza w stylu C - wskaźnik jest adresem w pamięci (liczba) i może być rzutowany na dowolny typ (również błędnie)

wskaźniki

+ efektywne
+ wygodne, zwłaszcza w stylu C - wskaźnik jest adresem w pamięci (liczba) i może być rzutowany na dowolny typ (również błędnie)
- niebezpieczne (zwłaszcza przy rzutowaniu)
- złożona arytmetyka
- problem właściciela obiektu - kto jest odpowiedzialny za usunięcie obiektu i zwolnienie przydzielonej pamięci?

Język C++ implementuje dwa operatory charakterystyczne dla arytmetyki wskaźników, które mogą być implementowane przez użytkownika:

- operator dereferenncji (wyłuskania obiektu wskazywanego przez wskaźnik)

```
R& K::operator*(void);
```

Język C++ implementuje dwa operatory charakterystyczne dla arytmetyki wskaźników, które mogą być implementowane przez użytkownika:

- operator dereferenncji (wyłuskania obiektu wskazywanego przez wskaźnik)

```
R& K::operator*(void);
```

- operator adresowania pośredniego (wyłuskanie składowej obiektu wskazywanego przez wskaźnik)

```
R* K::operator->(void);
```

Typ zwracany przez operator dereferencji musi być typem na który wywołanie operatora dereferencji (->) ma sens. Może to być wskaźnik lub inny obiekt klasy dla której dany operator został zaimplementowany.
Wywołanie operatora dereferencji na obiekcie powoduje wywołanie sekwencji tych operatorów na wszystkich zwracanych obiektach.
Jeżeli obiekt x jest typu X, który implementuje `operator->()` poniższa instrukcja:

```
x->y
```

tożsama jest instrukcji:

```
x.operator->()->y
```

Dzięki zaimplementowaniu operatorów wyłuskania istnieje możliwość stworzenia klasy, której obiekty będą zachowywały się ja wskaźniki:

```
template<typename T>
class my_ptr{
public:
    my_ptr(T* prt);
    ~my_ptr(void);

    T& operator*(void);
    T* operator->(void);

private:
    void clean(void);
    T* ptr_ = nullptr;
};


int main(void){
    auto s1 = my_ptr<my_class>(new my_class(8));
    std::cout<< (*s1).value << std::endl;
    std::cout<< s1->value << std::endl;

    return 0;
}
```

Jaki jest sens istnienia w systemie takiego obiektu?

Jakie są wady przedstawionego rozwiązania?

- przede wszystkim nic nie zostało usprawnione
- nadal koniecznie jest zarządzanie pamięcią - tworzenie obiektów i ich niszczenie

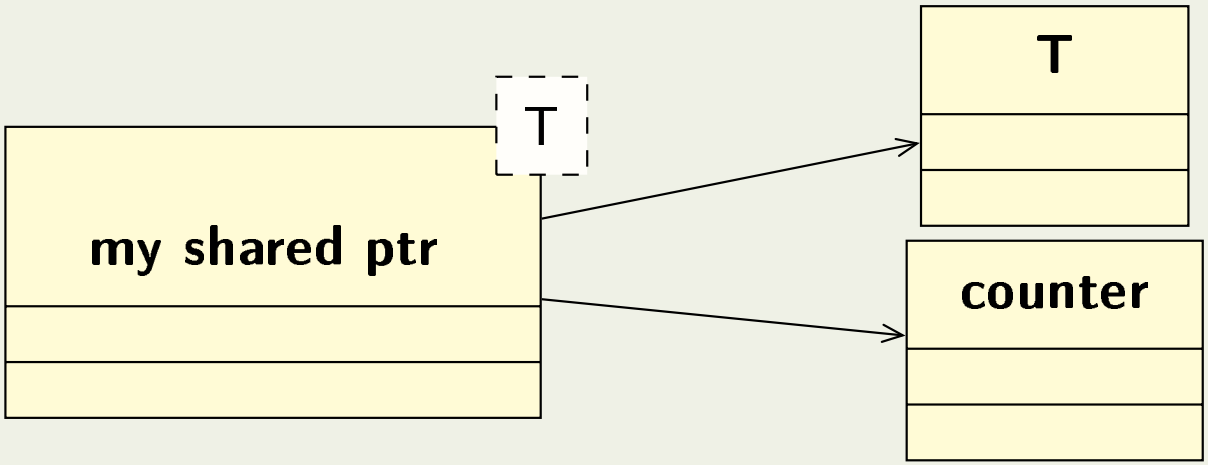
Jakie są wady przedstawionego rozwiązania?

- przede wszystkim nic nie zostało usprawnione
- nadal koniecznie jest zarządzanie pamięcią - tworzenie obiektów i ich niszczenie

Jakie są możliwości usprawnienia klasy `my_ptr`?
Przede wszystkim należy rozwiązać problemy związane z zarządzaniem pamięcią:

- kto jest właścicielem obiektu?
- kiedy obiekt jest niszczony?
- w jaki sposób obiekt jest niszczony?
- co jeżeli próbujemy odwołać się do obiektu który został zniszczony

niepełna specyfikacja za pomocą diagramu klas



niepełna specyfikacja za pomocą kodu

```
template<typename T>
class my_shared_ptr
{
public:
    /* ... */
private:
    T* ptr_ = nullptr;
    size_t* count_ = nullptr;
};
```

pełniejsza lecz wciąż niepełna specyfikacja za pomocą opisu

propozycja rozwiązania problemu oraz podjęte decyzje projektowe:

- obiekt tworzony jest przez specjalnie przygotowaną do tego funkcję
- alokacja pamięci następuje wewnątrz tej funkcji
- obiekty mogą być kopiowane oraz przenoszone (*co się dzieje w trakcie kopiowania, a co w trakcie przenoszenia?*)
- kopiowanie powoduje powołanie nowego wskaźnika wskazującego na istniejący już obiekt
- niszczenie obiektu nie powoduje niszczenia wskaźnika, o ile inne obiekty wskazują na obiekt wskazywany
- w tym celu tworzony jest specjalny licznik, który zlicza ilość wskazań na dany obiekt
- obiekt jest niszczony, gdy liczba wskazań jest równa zeru

specyfikacja za pomocą kodu

```
template<typename T>
class my_shared_ptr
{
public:
    my_shared_ptr(void);
    my_shared_ptr(const my_shared_ptr&  shared);
    my_shared_ptr(      my_shared_ptr&& shared);
    ~my_shared_ptr(void);

    T& operator*(void);
    T* operator->(void);

    my_shared_ptr& operator=(const my_shared_ptr&  shared);
    my_shared_ptr& operator=(      my_shared_ptr&& shared);

    template<typename T, typename... Args>
    friend my_shared_ptr<T> make_my_shared_ptr(Args&&... args);

    size_t count(void);

private:
    my_shared_ptr(T* prt);
    void clean(void);

    T* ptr_ = nullptr;
    size_t* count_ = nullptr;
};
```

funkcja alokująca pamięć

```
template<typename T, typename... Args>
my_shared_ptr<T> make_my_shared_ptr(Args&&... args)
{
//     return my_shared_ptr<T>(new T(args...));
    return my_shared_ptr<T>(new T(std::forward<Args>(args)...));
}
```

funkcja alokująca pamięć

```
template<typename T, typename... Args>
my_shared_ptr<T> make_my_shared_ptr(Args&&... args)
{
//     return my_shared_ptr<T>(new T(args...));
    return my_shared_ptr<T>(new T(std::forward<Args>(args)...));
}
```

konstruktory

```
template<typename T>
my_shared_ptr<T>::my_shared_ptr(void) {}

template<typename T>
my_shared_ptr<T>::my_shared_ptr(T* ptr)
    : ptr_(ptr), count_(new size_t(1))
{
}
```

destruktor

```
template<typename T>
my_shared_ptr<T>::~my_shared_ptr(void)
{
    clean();
}

template<typename T>
void my_shared_ptr<T>::clean(void)
{
    if (nullptr == count_) return;

    (*count_)--;
    if (0 == *count_)
    {
        delete ptr_;
        delete count_;
    }
    ptr_ = nullptr;
    count_ = nullptr;
}
```

konstruktor kopiujący i przenoszący

```
template<typename T>
my_shared_ptr<T>::my_shared_ptr(const my_shared_ptr & shared)
    : ptr_(shared.ptr_), count_(shared.count_)
{
    (*count_)++;
}

template<typename T>
my_shared_ptr<T>::my_shared_ptr(my_shared_ptr && shared)
    : ptr_(shared.ptr_), count_(shared.count_)
{
    shared.ptr_ = nullptr;
    shared.count_ = nullptr;
}
```

konstruktor kopiujący i przenoszący

```
template<typename T>
my_shared_ptr<T> &
my_shared_ptr<T>::operator=(const my_shared_ptr<T> & shared){
    clean();
    ptr_ = shared.ptr_;
    count_ = shared.count_;
    (*count_)++;
    return *this;
}

template<typename T>
my_shared_ptr<T> &
my_shared_ptr<T>::operator=(my_shared_ptr<T> && shared){
    clean();
    ptr_ = shared.ptr_;
    count_ = shared.count_;
    shared.ptr_ = nullptr;
    shared.count_ = nullptr;
    return *this;
}
```

operatory wyłuskania

```
template<typename T>
T & my_shared_ptr<T>::operator*(void){
    return *ptr_;
}

template<typename T>
T* my_shared_ptr<T>::operator->(void){
    return ptr_;
}
```

operatory wyłuskania

```
template<typename T>
T & my_shared_ptr<T>::operator*(void){
    return *ptr_;
}

template<typename T>
T* my_shared_ptr<T>::operator->(void){
    return ptr_;
}
```

statystyka

```
template<typename T>
size_t my_shared_ptr<T>::count(void) {
    if (nullptr == count_) return 0;

    return *count_;
}
```

test

```
struct my_class{
    int value = 0;
    my_class(int value) : value(value) {}
    my_class(int value1, int value2) : value(value1 + value2) {}
};

int main(void)
{
    auto s1 = make_my_shared_ptr<my_class>(8);
    auto s2 = make_my_shared_ptr<my_class>(4, 5);
    auto s3 = s1;

    std::cout << s1.count() << " "; // ... << s2.count()
    std::cout << s1->value << " ";  // ... << s2->value

    s1 = s2;
    std::cout << s1.count() << " "; // ... << s2.count()
    std::cout << s1->value << " ";  // ... << s2->value

    s1 = std::move(s3);
    std::cout << s1.count() << " "; // ... << s2.count()
    std::cout << s1->value << " ";  // ... << s2->value
    return 0;
}
```

W jaki sposób zapewnić pojedynczą instancję wskaźnika ?

W jaki sposób zapewnić pojedynczą instancję wskaźnika ?

- należy zabronić kopiowania (konstruktor i operator)

W jaki sposób zapewnić pojedynczą instancję wskaźnika ?

- należy zabronić kopiowania (konstruktor i operator)
- a co należy zrobić z przenoszeniem ?

W jaki sposób zapewnić pojedynczą instancję wskaźnika ?

- należy zabronić kopiowania (konstruktor i operator)
- a co należy zrobić z przenoszeniem ?

```
template<typename T>
class my_unique_ptr{
public:
    my_unique_ptr(void) = delete;
    my_unique_ptr(const my_shared_ptr&  shared) = delete;
    my_unique_ptr(      my_shared_ptr&& shared);
    ~my_unique_ptr(void);

    T& operator*(void);
    T* operator->(void);

    my_unique_ptr& operator=(const my_unique_ptr&  shared) = delete;
    my_unique_ptr& operator=(      my_unique_ptr&& shared);

    template<typename T, typename... Args>
    friend my_unique_ptr<T> make_my_unique_ptr(Args ... args);
private:
    my_unique_ptr(T* prt);
    void clean(void);
    T* ptr_ = nullptr;
    size_t* count_;  };
```

Biblioteka standardowa języka C++ w nagłówku `memory` udostępnia sprytne wskaźniki o zachowaniu zbliżonym do klas przedstawionych na wykładzie. W skład tego modułu wchodzą:

- wskaźniki współdzielone - `std::shared_ptr`
- wskaźniki unikalne - `std::unique_ptr`
- wskaźniki słabe - `std::weak_ptr`

Każdy z tych wskaźników ma swoje zastosowania w projektowanych systemach. Słabe wskaźniki nie zarządzają przekazaną pamięcią a jedynie pozwalają z niej korzystać. Przed wyłuskaniem słabych wskaźników należy sprawdzić czy wskazywany obiekt istnieje.

https://en.cppreference.com/w/cpp/memory