Programowanie Obiektowe

Marcin Kamil Bączyk

Wykład 7

14 stycznia 2021

Treść dzisiejszego wykładu

- obiekty funkcyjne
- `std::function`
- algorytmy z biblioteki standardowej

wskaźnik i referencja do funcji

- zmienna przechowująca adres funkcji, która może być później wywołana
- niskopoziomowa funkcjonalność języka C (umożliwia implementację polimorfizmu dynamicznego)
- może być wykorzystana do łączenia z niskopoziomowym kodem z języka C
- należy unikać w projektach C++

deklaracja wskaźnika do funkcji

```
int fun(int, bool);
using fun_ptr = decltype(fun)*;
```

definicje funkcji

```
int fun_1(int a, bool b)
{
    return b ? a : 0;
}

int fun_2(int a, bool b)
{
    return a + b;
}
```

wykorzystanie wskaźników i referencji

```
int main()
{
    fun_ptr fp;

    fp = &fun_1;

    std::cout << "fp(2, true) = " << fp(2, true) << '\n';
    std::cout << "fp(2, false) = " << fp(2, false) << '\n';

    fp = &fun_2;

    std::cout << "fp(2, true) = " << fp(2, true) << '\n';
    std::cout << "fp(2, false) = " << fp(2, false) << '\n';

    auto& fr = fun_1;
    std::cout << "fr(2, false) = " << fr(2, false) << '\n';
    // fr = fun_2;  blad: assignment of read-only reference

    return 0;
}
```

klasy z przeciążonym operatorem wywołania

- obiektowy sposób do tworzenia typów zachowujących się jak funkcje
- implementacja klasy jest jedynie opisem sposobu działania „funkcji”
- bytem wywoływalnym jest konkretna instancja klasy - obiekt
- obiekt może mieć swój wewnętrzny stan
- stan może być mutowalny bądź nie

problem 1

Dane są obiekty reprezentujące osobę. Obiekty posiadają metodę zwracającą wiek osoby w danym roku. Należy przygotować funkcję, która będzie sprawdzała czy dana osoba jest pełnoletnia.

definicja klasy

```
class Person
{
public:
    Person(std::string name, int birthDate)
    : name_(name), birthDate_(birthDate) {}

    auto name() const { return name_; }
    auto age(int when) const {return when - birthDate_;}

private:
    std::string name_;
    int birthDate_;
};
```

rozwiązanie naiwne

```
bool olderThan18(const Person& person, int when)
{
    return person.age(when) > 18;
}
```

A co w przypadku gdy pełnoletność osiągana jest w innym wieku ?

rozwiązanie naiwne

```
bool olderThan18(const Person& person, int when)
{
    return person.age(when) > 18;
}
```

A co w przypadku gdy pełnoletność osiągana jest w innym wieku ?

wykorzystanie innej funkcji - ??

```
bool olderThan21(const Person& person, int when)
{
    return person.age(when) > 21;
}
```

klasa z operatorem wywołania

```
class OlderThan
{
public:
    OlderThan(int age, int when) : age_(age), when_(when) {}

    bool operator()(const Person& person)
    {
        return person.age(when_) > age_;
    }
private:
    int age_;
    int when_;
};
```

wykorzystanie

```
int main()
{
    auto p1 = Person("Jan Kowalski", 1995);
    auto p2 = Person("Julia Nowak", 2004);
    std::cout << olderThan18(p1,2021) << '\n';
    std::cout << olderThan18(p2,2021) << '\n';

    auto olderThan18 = OlderThan(18, 2021);
    auto olderThan35 = OlderThan(35, 2021);

    std::cout << olderThan18(p1) << '\n';
    std::cout << olderThan35(p1) << '\n';

    return 0;
}
```

problem 2

Dane są obiekty reprezentujące samochód zawierającą metodę zwracającą wiek samochodu w danym roku. Należy przygotować funkcjonalność, sprawdzającą czy dany samochód jest zabytkiem.

definicja klasy

```
class Car
{
public:
    Car(std::string brand, int productionDate)
    : brand_(brand), productionDate_(productionDate) {}

    auto brand() const { return brand_; }
    auto age(int when) const {return when - productionDate_;}

private:
    std::string brand_;
    int productionDate_;
};
```

definicja klasy szablonowej

```
template<typename T>
class OlderThan
{
public:
    OlderThan(int age, int when) : age_(age), when_(when) {}

    bool operator()(const T& object)
    {
        return object.age(when_) > age_;
    }
private:
    int age_;
    int when_;
};
```

wykorzystanie obiektu funkcyjnego

```
auto olderThan18 = OlderThan<Person>(18, 2021);
auto olderThan25 = OlderThan<Car>(25, 2021);

auto c1 = Car("Ford Focus", 2015);
auto c2 = Car("Mercedes-Benz 190 SL", 1960);

// std::cout << olderThan18(c1) << '\n'; blad !!!
std::cout << olderThan25(c1) << '\n';
std::cout << olderThan25(c2) << '\n';
```

Pojęcie wieku nie zależy od typu obiektu. Nie należy się przywiązywać do konretnego typu tam gdzie nie jest to konieczne.

definicja klasy z szablonowym operatorem wywołania

```
class OlderThan
{
public:
    OlderThan(int age, int when) : age_(age), when_(when) {}

    template<typename T>
    bool operator()(const T& object)
    {
        return object.age(when_) > age_;
    }
private:
    int age_;
    int when_;
};
```

wykorzystanie obiektu funkcyjnego

```
auto olderThan18 = OlderThan(18, 2021);
auto olderThan35 = OlderThan(35, 2021);
auto olderThan25 = OlderThan(25, 2021);

auto c1 = Car("Ford Focus", 2015);
auto c2 = Car("Mercedes-Benz 190 SL", 1960);

std::cout << olderThan18(c1) << '\n';
std::cout << olderThan25(c1) << '\n';
std::cout << olderThan25(c2) << '\n';
```

wyrażenia lambda

- umozliwiają tworzenie obiektów funkcyjnych w miejscu
- pozwalają unikania tworzenia klas lub funkcji tam gdzie nie jest to potrzebne
- wyrażenia mają swój typ - są to klasy
- typ wyrażenia lambda jest nieokreślony, nie można się nim posłużyć do utworzenia innego obiektu
- mają semantykę prawych wartości
- lukier składniowy

wyrażenie lambda

```
auto olderThan21 =
    [when = 2021, limit = 21](const Person& person)
    {return person.age(when) > limit;};

std::cout << olderThan21(p1) << '\n';
// std::cout << olderThan21(c1) << '\n'; blad !!!
```

wyrażenie lambda

```
auto olderThan21 =
    [when = 2021, limit = 21](const Person& person)
    {return person.age(when) > limit;};

std::cout << olderThan21(p1) << '\n';
// std::cout << olderThan21(c1) << '\n'; blad !!!
```

uogólnione wyrażenie lambda

```
auto olderThan5 =
    [when = 2021, limit = 21](const auto& object)
    {return object.age(when) > limit;};

std::cout << olderThan5(p1) << '\n';
std::cout << olderThan5(c1) << '\n';
```

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

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.

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.

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 AgeChecker = std::function<bool(const T&)>;

int main(void)
{
    /* ... */

    std::vector<AgeChecker<Person>> age_checkers;
    age_checkers.push_back(olderThan18);
    age_checkers.push_back(olderThan21);

    for(auto& checker : age_checkers)
    {
        std::cout << checker(p1) << '\n';
    }

    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;});
```