Języki i metody programowania 2

Lab 10

Szablony

Przy pomocy szablonów możemy zdefiniować przepis na funkcję, albo klasę. Na podstawie tego szablonu kompilator generuje następnie właściwy kod metod pod każdy przypadek użycia napotkany w kodzie. Szablon metody IsLess:

```
template<class T>
bool IsLess(const T &left, const T &right) {
  return left < right;
}
```

To jest tylko przepis jak zrobić funkcję IsLess dla dowolnego typu, jedyny wymóg jaki jest to że ten typ musi mieć zdefiniowany **operator<**, bo taki jest wykorzystywany w kodzie metody. Niestety ponieważ IDE nie ma pojęcia co będzie w przyszłości podstawione pod typ T nie będzie nam podpowiadało jakie metody są dostępne dla tego typu.

Przykład użycia:

```
bool result = IsLess<int>(2, 7); //zostanie wygenerowana metoda IsLess(const int &left, const int &right);

bool result = IsLess<double>(8.019, 1.901); //j.w. ale IsLess(const double &left, const double &right);

bool result = IsLess<string>("abc", "efg"); //IsLess(const string &left, const string &right);

//można pominąć typ szablonu i pozwolić kompilatorowi się domyśleć
//o jaką wersje metody nam chodzi:
bool result = IsLess(55, 99);
bool result = IsLess(StudentId {4}, StudentId{7}); //będzie działać pod warunkiem, że klasa StudentId ma przeładowany operator<
```

Zadania

1. Przygotuj szablon Repository, który udostępnia metody
   a. Size – zwraca ilość elementów w repozytorium
   b. operator[] – zwracający wskazany element z tablicy
   c. FindBy(Query) – zwracający elementy repozytorium dla dowolnego (jednego) Query

   W metodzie main wykorzystaj szablon dla typów Employee oraz Student.

```cpp
#include <cstddef>
#include <initializer_list>
#include <algorithm>

template<class T>
class MyVector {
 public:
  MyVector(size_t intial_capacity=16);
  MyVector(const std::initializer_list<T> &elements);
  ~MyVector();
  MyVector(const MyVector<T> &to_copy);
  MyVector(MyVector<T> &&to_move);
  void operator=(const MyVector<T> &to_copy);
  void operator=(MyVector<T> &&to_move);


  const T &operator[](size_t index) const;
  T &operator[](size_t index);
  size_t Size() const;
  void Reserve(size_t size);

  void PushBack(T i);
 private:
  void Swap(MyVector<T> &&other);
  void Copy(const MyVector<T> &other);
  void Destroy();


  T *elements_;
  size_t size_;
  size_t capacity_;
};

template<class T>
MyVector<T>::MyVector(size_t intial_capacity) {
  Reserve(intial_capacity);
}

template<class T>
MyVector<T>::MyVector(const std::initializer_list<T> &elements) {
  Reserve(elements.size());
  int i =0;
  for (const auto &el : elements) {
    elements_[i++] = el;
  }
  size_ = elements.size();
}

template<class T>
MyVector<T>::~MyVector() {
  Destroy();
}

template<class T>
MyVector<T>::MyVector(const MyVector<T> &to_copy) {
  Copy(to_copy);
}

template<class T>
MyVector<T>::MyVector(MyVector<T> &&to_move) {
  elements_= nullptr;
  Swap(std::move(to_move));
}

template<class T>
```

```
void MyVector<T>::operator=(const MyVector<T> &to_copy) {
  if (this != &to_copy) {
    Destroy();
    Copy(to_copy);
  }
}

template<class T>
void MyVector<T>::operator=(MyVector<T> &&to_move) {
  if (this != &to_move) {
    Destroy();
    Swap(std::move(to_move));
  }
}

template<class T>
const T &MyVector<T>::operator[](size_t index) const {
  return elements_[index];
}

template<class T>
T &MyVector<T>::operator[](size_t index) {
  return elements_[index];
}

template<class T>
size_t MyVector<T>::Size() const {
  return size_;
}

template<class T>
void MyVector<T>::Swap(MyVector<T> &&other) {
  std::swap(size_,other.size_);
  std::swap(capacity_,other.capacity_);
  std::swap(elements_,other.elements_);
}

template<class T>
void MyVector<T>::Destroy() {
  size_ = 0;
  capacity_ = 0;
  delete [] elements_;
  elements_ = nullptr;
}

template<class T>
void MyVector<T>::Reserve(size_t size) {
  elements_ = new T[size];
  capacity_ = size;
  size_ = 0;
}

template<class T>
void MyVector<T>::Copy(const MyVector<T> &other) {
  Reserve(other.size_);
  size_ = other.size_;
  for(int i=0; i<other.size_; ++i) {
    elements_[i] = other.elements_[i];
  }
}

template<class T>
void MyVector<T>::PushBack(T t) {
  if (size_ >= capacity_) {
    MyVector tmp(capacity_*2);
    for(int i=0; i<size_; ++i) {
```

```
      tmp.elements_[i] = elements_[i];
    }
    tmp.size_ = size_;
    *this = std::move(tmp);
  }
  elements_[size_++] = t;
}
```