Vježba L01 II

Jakov Spahija

17. svibnja 2021.

Sadržaj

1 Interfaces **2**

2 Operators **4**

3 Associations **6**

4 Templates **8**

5 Variadics **10**

6 Perfect Forwarding **12**

1. Interfaces

Sučelje je apstraktna klasa samo sa čistim virtualnim funkcijama, bez ikakvog alociranog objekta. Predstavlja zahtjeve koje treba ispuniti objekt koji će implementirati korisnički kod.

IAccount.h

```
5       struct IAccount {
6               virtual bool Deposit(double) = 0;
7               virtual bool Withdraw(double) = 0;
8               virtual void Balance(double) = 0;
9               virtual double Balance() const = 0;
10              ~IAccount();
11      };
```

Svaka klasa koja implementira neko sučelje, mora zadovoljavati potpis funkcije definiranog unutar sučelja.
Implementacija sučelja `IAccount` unutar druge datoteke:

Prijepis 1.1. Account.h

```
1   #pragma once
2   #include<iostream>
3
4   namespace fin {
5       class Account : public IAccount {
6       private:
7               double _balance;
8       public:
9               Account(double balance) :_balance{ balance } {};
10              virtual bool Deposit(double amount) override {
11                      _balance += amount;
12                      std::cout << "Deposit: " << amount << std::endl;
13                      return _balance;
14              }
15              virtual bool Withdraw(double amount) override {
16                      _balance -= amount;
17                      std::cout << "Withdraw: " << amount << std::endl;
18                      return _balance;
19              }
20              virtual void Balance(double balance) override {
21                      _balance = balance;
22                      std::cout << "New balance: " << balance << std::endl;
23              }
24              virtual double Balance() const override { return _balance; }
25      };
26  }
```

Terminal

```
Balance: 100
Deposit: 100
Withdraw: 100
Balance: 100
~IAccount
Balance: 1000
Deposit: 100
Withdraw: 100
Balance: 1000
~IAccount
```

2. Operators

*Smart pointer*i iz prve vježbe implementiramo pomoću *template* konstrukcija koja djeluje kao šablone i vežu za klasu, strukturu ili funckiju koja slijedi nakon definicije predloška.

Prijepis 2.1. Smart Pointer.h: Klasa

```
5   template<typename T>
6   class SmartPtr {
7   private:
8         T* _pointee;
9   public:
10        SmartPtr(T* pT = nullptr) : _pointee{ pT } {}
11        SmartPtr& operator=(SmartPtr&);
12        T const operator*() const;
13        T operator*();
14        T const* operator->() const;
15        T* operator->();
16        ~SmartPtr();
17  };
```

Kod ove implementacije smart pokazivača, *assignment* operator bi trebao 'ukrasti' na ono što pokazuje smart pokazivač sa desne strane jednakosti, efektivno koristeći *move* semantiku.

operator=()

```
19  template<typename T>
20  SmartPtr<T>& SmartPtr<T>::operator=(SmartPtr<T>& p) {
21        std::cout << "SmartPtr::operator=()" << std::endl;
22        if (this != &p) {
23              delete(_pointee);
24              _pointee = std::move(p._pointee);
25              p._pointee = nullptr;
26        }
27        return *this;
28  }
```

Dereference i *arrow* operatori imaju svoje **const** verzije funkcija, kao što je navedeno u Prijepisu 2.1.

operator*()

```
36  template<typename T>
37  T SmartPtr<T>::operator*() {
38        std::cout << "SmartPtr::operator*()" << std::endl;
39        return *_pointee;
40  }
```

operator->()

```
48        template<typename T>
49        T* SmartPtr<T>::operator->() {
50              std::cout << "SmartPtr::operator->()" << std::endl;
51              return _pointee;
52        }
```

- **What would happen if you were to make another call to function m on sp?**

Zbog definicije *assignment* operatora koji *move* semantiku, sp pokazivaču je ukraden resurs na koji je pokazivao, te je pozvan destruktor za njega.

Program.cpp

```
23              sq = sp;
24              sq->m();
25              // sp->m(); Read access violation - null pointer assigned
```

Terminal

```
Something konstruktor: 1
Something konstruktor: 2
Something konstruktor: 10
Something konstruktor: 20

SmartPtr::operator->()
vrijednost: 10

(*sq).m()
SmartPtr::operator*()
vrijednost: 20
Something destruktor 20

sp = sq
SmartPtr::operator=()
Something destruktor 20
SmartPtr::operator->()
vrijednost: 10
Something destruktor 10
```

3. Associations

U ovom zadatku promatramo dvije asocijacije između `Boss` i `Mafia` klasa, te `Date` i `Person` klasa.

Boss.h

```
7       class Boss {
8       private:
9             std::wstring _name;
10            Mafia* _mafia = nullptr;
```

Prijepis 3.1. Boss.h: Klasa Boss, veza na mafiju

```
18            void SetMafia(Mafia* mafia) {
19                  _mafia = mafia;
20                  // std::wstring << "Boss -> " << mafia->ToString() <<
                        std::endl;
21            }
22            virtual void Order(void) {
23                  _mafia->DoIt();
24            }
```

Prijepis 3.2. Date.h

```
1   #pragma once
2   #include<iostream>
3   #include<string>
4
5   namespace abc {
6         struct Date {
7               int day, month, year;
8               Date(int d, int m, int y) : day{ d }, month{ m }, year{ y } {
9                   std::cout << "Konstruktor Date(): " << d << " / " << m
                        << " / " << y << std::endl;
10              }
11        };
12  }
```

Prijepis 3.3. Person.h

```
1   #pragma once
2   #include<iostream>
3   #include"Date.h"
4   #include<string>
5
6   namespace abc {
```

```
7         struct Person {
8               std::wstring _name;
9               const Date _birhtday;
10              Person(std::wstring name, const Date birthday) : _name{ name },
                     _birhtday{ birthday }{
11                    std::wcout << L"Konstruktor Person(): " << _name << std
                          ::endl;
12              }
13              ~Person() {
14                    std::wcout << L"Dekonstruktor ~Person(): " << _name <<
                          std::endl;
15              }
16        };
17  }
```

Terminal

```
Konstruktor Boss(): Paulie Walnuts
Konstruktor Mafia(): Naša
Buisness: Igračke
Dekonstruktor ~Mafia(): Naša
Konstruktor Date(): 14 | 2 | 1988
Konstruktor Person(): Tony Baloney
Dekonstruktor ~Person(): Tony Baloney
Dekonstruktor ~Boss(): Paulie Walnuts
```

4. Templates

Template je predložak ili šablona za klasu, strukturu ili funkciju. Može sadržavati *type* parametar (**typename**), običan parametar ili sam template. Kompajer generira kod tijekom *compile time* ovisno o tome koji su argumenti proslijeđenit tijekom korištenja koda.

Cilj zadataka je implementirati stog, koji može biti definiran za bilo koji tip, koristeći **template**.

Prijepis 4.1. Stack klasa

```
6  namespace tpl {
7        template<typename T, size_t N>
8        class Stack {
9        private:
10             std::array<T, N> _elements;
11             size_t _count;
12       public:
13             Stack() :_count{ 0 }, _elements() {};
14             void Push(const T&);
15             void Pop();
16             const T& Top() const;
17             bool Empty() const {
18                   return _count; // size_t jest >= 0
19             }
20             size_t Size() const { return _count; }
21       };
```

Prijepis 4.2. Stack.h: Primjer šablone za funkciju

```
39       template<typename T, size_t N>
40       inline void Show(Stack<T, N> stack, int size) {
41             while (stack.Size()) {
42                   std::wcout << stack.Top() << L" ";
43                   stack.Pop();
44             }
45             std::cout << std::endl;
46       }
```

Variadic template sadržava parametre koji koriste *rest* operator ..., a ako se žele proširiti u normalni oblik, *rest* operator se stavi kao sufiks na identifikator.

Prijepis 4.3. Variadic.h

```
1  #pragma once
2  #include<iostream>
3
4  namespace tpl {
5         void Print() {
6                std::wcout << std::endl;
7         }
8
9         template<typename T, typename... Ts>
10        inline void Print(T head, Ts... rest) {
11               std::wcout << head << L" ";
12               Print(rest...);
13        }
14 }
```

Terminal

```
2
b
7 6 5 4 3 2 1 0
a a a
1
1 a
a 1 1 b
```

5. Variadics

Prijepis 5.1. Templates.h

```
#pragma once
#include<windows.h>
#include<string>

namespace vtl {

#ifdef DEBUG
	void Log() {
		OutputDebugStringW(L"\n");
	}

	template<typename T, typename... Ts>
	inline void Log(const T& head, const Ts&... tail) {
		OutputDebugStringW(head);
		OutputDebugStringW(L" ");
		Log(tail...);
	}

	template<typename T, typename... Ts>
	inline void Log(const T& head, const Ts&... tail, int a, int b) {
		if (a && b)
			Log(head, tail...);

	}
#else
#define Log(x, ...);
#endif
}
```

Prijepis 5.2. Program.cpp

```
#include<iostream>

#define DEBUG
#include"Templates.h"
using namespace std;
using namespace vtl;
using namespace std::string_literals;

int main() {

	Log(L"a", L"b");
	// Log(L"a", L"b", 1, 1);

	return 0;
}
```

Debug Window

```
'Variadics.exe' (Win32): Loaded 'C:\Users\Jakov\FESB\2. semestar\3D
    Simulacije\Izvještaji\vjezba2\Debug\Variadics.exe'. Symbols loaded.
'Variadics.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\SysWOW64\vcruntime140d.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ucrtbased.dll'.
The thread 0x34ec has exited with code 0 (0x0).
a b
'Variadics.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel.appcore.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcrt.dll'.
'Variadics.exe' (Win32): Loaded 'C:\Windows\SysWOW64\rpcrt4.dll'.
The thread 0x3c08 has exited with code 0 (0x0).
The thread 0x29ec has exited with code 0 (0x0).
The program '[1696] Variadics.exe' has exited with code 0 (0x0).
```

6. Perfect Forwarding

S obzirom da izrazi imaju i svoju vrijednosnu kategoriju, da bi se pravilno proslijedila kategorija preko *template* argumenta, koristimo koncept *universal reference*. Univerzalna referenca je parametar ili varijabla koja ima kao za svoj tip `T&&`.
Pravila sažimanja reference:

1. `T& &` postane `T&`
2. `T& &&` postane `T&`
3. `T&& &` postane `T&`
4. `T&& &&` postane `T&&`

Pravilo kod dedukcije *template* argumenata:

```
1  template<typename T>
2        void foo(T&&);
```

1. Kad se `foo` proslijedi *lvalue* tipa `A`, tad se `T` zamjeni sa `A&`, a nakon primjene pravila sažimanja postane `A&`
2. Kad se `foo` proslijedi *rvalue* tipa `A`, tad se `T` zamjeni sa `A` a argument postane `A&&`

`std::forward` uspješno prosljeđuje vrijednosnu kategoriju izraza, za razliku od `std::move` koji bezuvjetno pretvara izraz iz *lvalue* kategoriju u *rvalue*.

std::forward

```
1        template<class S>
2        S&& forward(typename remove_reference<S>::type& a) noexcept{
3              return static_cast<S&&>(a);
4        }
```

std::move

```
1        template<class T>
2        typename remove_reference<T>::type&&
3        std::move(T&& a) noexcept{
4              typedef typename remove_reference<T>::type&& RvalRef;
5              return static_cast<RvalRef>(a);
6        }
```

Prijepis 6.1. Functions.h: perfect forwarding

```
7         void f(int& lref) {
8               std::wcout << L"Prenesena vrijednost: " << lref << std::endl;
9         }
10        void f(int&& lref) {
11              std::wcout << L"Prenesena vrijednost: " << lref << std::endl;
12        }
13
14        template<typename T>
15        void Forward(T&& value) {
16              f(std::forward<int&&>(value));
17        }
```

Prijepis 6.2. Functions.h: Variadic primjer

```
19        template<typename T, typename ...Ts>
20        inline mem::SmartPtr<T> MakeSmart(Ts&&... rest){
21              std::cout << "Broj parametara: " << sizeof...(rest) << std::
                    endl;
22              return mem::SmartPtr<T>{new T{ std::forward<Ts>(rest)... }};
23        }
```

Terminal

```
Prenesena vrijednost: 1
Prenesena vrijednost: 2
Prenesena vrijednost: 3
Prenesena vrijednost: 1
Prenesena vrijednost: 2

Broj parametara: 1
Konstruktor Something()
SmartPtr::operator->()
Vrijednost: 10

Broj parametara: 3
Konstruktor Nešto()
SmartPtr::operator->()
SmartPtr::operator->()
SmartPtr::operator->()
Vrijednost: 10 29.98 abc
Destruktor ~Nešto()
Destrkutor ~Something()
```