

Vježba 1

Jakov Spahija

22. travnja 2021.

Sadržaj

1	Tipovi i objekti	2
2	Reference	5
3	Semantika	8
4	Smart Pokazivači	10
5	Nasljeđivanje	12

1. Tipovi i objekti

U uvodnoj vježbi, izvide se objektno orijentirane mogućnosti C++. Deklaracija te overload-nje funkcija, klasa.

U header datoteci `Circle.h` definiran je objekt `Circle` koji se sastoji of:

- član **double** `_radius`
- konstruktorom koji postavlja jedinstvenu varijablu tipa **double** na `1.0`
- konstantna, virtualna member funkcija `Area` koja vraća **double**
- konstantna, virtualna member funkcija `ToString` koja vraća **wstring**
- virtualni destruktork

Prijepis 1.1. Circle.h

```
1  #pragma once
2  #include<string>
3
4  namespace abc{
5      class Circle { // hello LaTeX
6      public:
7          double _radius;
8          Circle(double=1.0);
9          virtual double Area() const;
10         virtual std::wstring ToString() const;
11         virtual ~Circle();
12     };
13
14     Circle f(Circle C);
15     void g(Circle* C);
16     Circle* h(void);
17 }
```

Također su deklarirane i 3 funkcije `f`, `g` i `h`.

Definicija virtualnih funkcija klase, te ostale funkcije unutar **namespace** `abc`, se odvija kasnije u datoteci `Circle.cpp`.

```

Circle.cpp

10 Circle::Circle(double radius) :
11     _radius{ radius }
12 {
13     wcout << ToString() << endl;
14 }
15
16 double Circle::Area() const {
17     return _radius * _radius * PI;
18 }
19
20 wstring Circle::ToString() const {
21     return L"Circle, radius: " + to_wstring(_radius) + L"\nPovršine: " +
22         to_wstring(Area());
23 }
24 Circle::~Circle() {

```

```

Circle.cpp

28 Circle abc::f(Circle C) {
29     Circle newCircle = Circle((double)(C._radius + 10.0));
30     return newCircle;
31 }
32
33
34 void abc::g(Circle* C) {
35     C->_radius += 100;
36     wcout << to_wstring(C->_radius) << endl;
37 }
38
39 Circle* abc::h() {
40     Circle* newCircle = new Circle(0);
41     return newCircle;
42 }

```

What would happen if you were to create a non-dynamic object inside the function and return its address?

Životni vijek podataka deklariranih na stogu unutar nekog scope-a, traje za sve izraze unutar tog scope-a. Takav objekt će biti uništen/izbrisan.

```
Circle, radijus: 1.000000 Površine: 3.141593
3.141593
Circle, radijus: 1.000000 Površine: 3.141593
Unesi novi radijus: 23
Circle, radijus: 23.000000 Površine: 1661.902514
Desktruktor je pozvan
Circle, radijus: 23.000000 Površine: 1661.902514
Circle, radijus: 11.000000 Površine: 380.132711
Desktruktor je pozvan
Desktruktor je pozvan
površina od c: 380.133
Circle, radijus: 3.000000 Površine: 28.274334
103.000000
Desktruktor je pozvan
Circle, radijus: 0.000000 Površine: 0.000000
0
Desktruktor je pozvan
Desktruktor je pozvan
Desktruktor je pozvan
Desktruktor je pozvan
```

2. Reference

Izraz u C++ se karakterizira sa tipom i kategorijom vrijednosti. *Rvalue*, *lvalue* su nam najkorisnije kategorije za promatranje, dok su *xvalue*(*expiring*), *glvalue*(*global left*), *prvalue*(*pure right*) primitivne kategorije.

- *xvalue* je tranzicijska kategorija vrijednosti, koja služi u produživanju životnog vijeka resursa. `std::move` vraća takav izraz.
- *lvalue/rvalue* je *glvalue/prvalue* koji može biti i *xvalue*, respektivno

▪ What would happen if you were to increment the reference's value?

Lijeva referenca je u ovom slučaju samo referenca koji nije promjenjiva.

```
Something.h
19 void f(const Something& S) {
20     // S.value++; const lvalue ref
21     std::cout << "Vrijednost lref: " << S.value << std::endl;
22 }
```

▪ What would happen if you were to add another pointer to int and assign to it the address of expression 100? And if the pointer was a read-only pointer?

Takav izraz nije *lvalue*, a ako je izraz read-only pokazivač onda se može pridjeliti

```
Program.cpp
16 int* pi = (int* const)100; // ok
```

▪ What would happen if you tried to assign something else to l?

Efektivno se pridjeli ta vrijednost, onoj *lvalue* na koju se *l* referencira, u ovom slučaju *v*

▪ What would happen if you tried assigning v to a constant l-value reference to int?

Ako pridjelimo *v* nekakvoj `const int&`, preko takve se reference neće moći mijenjati.

▪ Increment l and output v's value to the console. Can you assign 100 to an l-value reference to int?

```
Program.cpp
40 // lvalue referenca se ne može inicijalizirati sa konstantom
41 // int& li = 100;
```

- Add a constant l-value reference `cl` to 100. Can you increment `cl`?

`cl` nije definirana kao referenca preko koje možemo modificirati.

- What would have happened had you uncommented the overloaded function prior to making calls to `h`?

Tijekom *compile time*, kompajler odredi koju funkciju poziva. Pod *overload resolution* kriterijima, spadaju i kategorije vrijednosti parametara. Ako je na jedini parametar proslijeđen *lvalue* izraz onda se preferira funkcija sa definicijom parametra *lvalue* reference (`const T&`). Također vrijedi isto i za *rvalue* izraz te referencu (`T&&`).

```
Something.h
29     void h(const Something& S) {
30         std::cout << "Vrijednost clref: " << S.value << std::endl;
31     }
32
33     void h(Something&& S) {
34         S.value++;
35         std::cout << "Vrijednost rref: " << S.value << std::endl;
36     }
```

```
v:2
v:3

l = u; u++;
v:1
l:1

v = cl
v:40

l++
v:41
l = 100

r++
r:101

ri2++
ri2:81

ri++
ri:82
Konstruktor je pozvan
Vrijednost lref: 2
Konstruktor je pozvan
Vrijednost lref: 10
Destruktor je pozvan
Konstruktor je pozvan
Vrijednost rref: 11
Destruktor je pozvan
Vrijednost rref: 3
Vrijednost clref: 3
Konstruktor je pozvan
Vrijednost rref: 21
Destruktor je pozvan
Vrijednost rref: 4
Destruktor je pozvan
```

3. Semantika

Copy i *Move* konstruktori (CC, MC), su inače implicitno definirani od strane kompajlera. Oni se pozivaju kad se objekt inicijalizira. *Assignment* semantika specificira kako će se objektu, koji je već inicijaliziran, pridjeliti neka vrijednost.

Ovakva konfiguracija konstruktora je postavljena da bi se, kad je predana kao argument, *rvalue* referenca, primjeni *move* semantika, koja efektivno 'krade' resurse of objekta na koji referenca pokazuje. Inače, *copy* semantika se primjenjuje u slučaju *lvalue* reference.

Overload resolution odabire poziv funkcija na takav način.

Something.h	
9	<code>/*CC*/ Something(const Something&);</code>
10	<code>/*MC*/ Something(Something&&) noexcept;</code>
11	<code>/*CAO*/ Something& operator = (const Something&);</code>
12	<code>/*MAO*/ Something& operator = (Something&&) noexcept;</code>

Prijepis 3.11. Something.cpp Implementacija CC,MC,CAO,MAO	
1	<code>#include<iostream></code>
2	<code>#include"Something.h"</code>
3	
4	<code>using namespace std;</code>
5	<code>using namespace abc;</code>
6	
7	<code>Something::Something(int i): _ptr{ new int{i} } {</code>
8	<code> cout << *_ptr << endl;</code>
9	<code>}</code>
10	
11	<code>Something::Something(const Something& S) : _ptr{ S._ptr } {</code>
12	<code> cout << "CC" << endl;</code>
13	<code>}</code>
14	
15	<code>Something::Something(Something&& S) noexcept : _ptr{move(S._ptr)} {</code>
16	<code> cout << "MC" << endl;</code>
17	<code> S._ptr = nullptr;</code>
18	<code>}</code>
19	
20	
21	<code>Something& Something::operator = (const Something& S) {</code>
22	<code> cout << "CAO" << endl;</code>
23	<code> if (this != &S) {</code>
24	<code> this->_ptr = S._ptr;</code>
25	<code> }</code>
26	<code> return *this;</code>
27	<code>}</code>
28	
29	<code>Something& Something::operator = (Something&& S) noexcept {</code>
30	<code> cout << "MAO" << endl;</code>
31	<code> if (this != &S) {</code>
32	<code> _ptr = move(S._ptr);</code>
33	<code>}</code>


```

34     S._ptr = nullptr;
35     return *this;
36 }
37
38 Something::~~Something(){
39     if (_ptr) {
40         cout << "Destruktor je pozvan " << *_ptr << endl;
41     } else {
42         cout << "Destruktor je pozvan " << 0 << endl;
43     }
44
45     delete _ptr;
46 }

```

Terminal

```

1
2
3

CC
CAO
MC
MAO

Destruktor je pozvan 1
Destruktor je pozvan 0
Destruktor je pozvan 0
Destruktor je pozvan 2
Destruktor je pozvan 3

```

4. Smart Pokazivači

Smart pointer je klasa koja upravlja dinamički alociranim objektima(resursima) i pravilno ih oslobađa(briše).

Tri najzastupljenija *smart pointera* su *shared*, *unique* i *weak* pokazivači.

- *Shared* - održava pristup i zajedničko vlasništvo nad objektom preko pokazivača, tako da nekoliko *shared_ptr* pokazivača mogu imati vlasništvo nad istim objektom. Takav objekt je oslobođen tek ako su svi *shared_ptr* uništeni, ili nakon što zadnjem preostalom *shared_ptr* pridjelimo neki drugi objekt. To je implementirano preko brojača - *reference counting*
- *Unique* - ima jedinstveno vlasništvo nad objektom, koji se oslobađa kad se *unique_ptr* uništi. Zbog toga što je jedinstven, primjenjuje se samo *move* semantika('krađa')
- *Weak* - samo pokazuje na objekt bez vlasništva nad njim, te se ne broji(*reference counting*).

```
Program.cpp
10 void main() {
11     SetUTF8(wcout);
12     auto s = make_shared<Something>(L"s");
13     wcout << s->Name() << L" ima " << to_wstring(s.use_count()) << endl;
14
15     auto s1{ s };
16     wcout << s1->Name() << L" ima " << to_wstring(s1.use_count()) << endl;
17     ;
18     auto s2{ s1 };
19     wcout << s2->Name() << L" ima " << to_wstring(s2.use_count()) << endl;
20     ;
21     weak_ptr<Something> w{ s2 };
22     wcout << L"Weak pointer ima " << to_wstring(w.use_count()) << endl;
23
24     auto u = make_unique<Something>(L"u");
25     wcout << (*u).Name() << endl;
26
27     auto u1{ move(u) };
28     wcout << (*u1).Name() << endl;
29
30     auto u2{ move(u1) };
31     wcout << (*u2).Name() << endl;
32
33     return;
34 }
```

- What would happen if you were to call function Name on w?

Weak pointer ne može pristupiti member funkcijama

- Cast u to an r-value and use it to initialize u1. Can you use u to initialize u1 without using `std::move`?

Može postojati samo jedan *unique pointer*, pa ga moramo "krasti".

- Call the function `Name` on `u2` and output the result. Can you call function `Name` on `u1`?

Ne, jer smo ukrali od `u1`, a jedinstven je samo `u2`.

Terminal
<pre>s ima 1 s ima 2 s ima 3 Weak pointer ima 3 u u u</pre>

5. Nasljeđivanje

Pomoću nasljeđivanja se kreira hijerarhija zaposlenika `Employee` koja služi kao baza za klase `Worker` i `Manager`. Unutar `Employee` imamo definiran kalup članskih funkcija pomoću `virtual` te `override` u izvedenim klasama.

```
Employee.h

6      class Employee {
7      private:
8          std::wstring _name;
9      public:
10         explicit Employee(std::wstring name) : _name{ name } {
11             std::wcout << _name << std::endl;
12         }
13
14         virtual double CalculatePay() {
15             return 0.0;
16         }
17
18         virtual std::wstring ToString() const {
19             return _name;
20         }
21         virtual ~Employee();
22     };
```

```
Worker.h

5      class Worker : public Employee
6      {
7      private:
8          double _wage;
9          int _hours;
10     public:
11         Worker(std::wstring name, double wage, int hours) :
12             Employee{ name },
13             _wage{ wage },
14             _hours{ hours }
15         {
16             std::wcout << L"Zaposlenik() " << ToString() << std::
17                 endl;
18         }
19         virtual double CalculatePay() override
20         {
21             return Employee::CalculatePay() + _wage * _hours;
22         }
23
24         virtual std::wstring ToString() const override
25         {
26             return Employee::ToString() + L", " + std::to_wstring(
27                 _wage * _hours);
28         }
29         virtual ~Worker();
```

```

30         {
31             std::wcout << L"~Radnik() " << ToString() << std::endl;
32         }
33     };

```

```

Manager.h
5     class Manager :public Employee {
6     private:
7         double _salary;
8     public:
9         Manager(std::wstring name, double salary) : Employee{ name },
10             _salary{ salary } {
11             std::wcout << L"Menadžer() " << ToString() << std::endl;
12         }
13         virtual double CalculatePay() override
14         {
15             return Employee::CalculatePay() + _salary;
16         }
17         virtual std::wstring ToString() const override
18         {
19             return Employee::ToString() + L", " + std::to_wstring(
20                 _salary);
21         }
22         virtual ~Manager()
23         {
24             std::wcout << L"~Menadžer() " << ToString() << std::endl;
25             ;
26         }
27     };

```

- Add a dynamic object of type `Employee` and assign it to a variable named `employee`. Can you assign `employee` to `manager`?

Klasa `Employee` je baza izvedene klase `Manager`

```

Manager.h
5     class Manager :public Employee {

```

- Delete the objects that pointers from the vector point to. What would happen if you tried to assign a wstring to a variable of type Employee?

Konstruktor za Employee je explicit

Employee.h	
10	explicit Employee(std::wstring name) : _name{ name } {
11	std::wcout << _name << std::endl;
12	}

Terminal
Šef
Menadžer() Šef,10000.000000
Tip: class com::Manager
 Ana
Zaposlenik() Ana, 17600.000000
Zoi
Zaposlenik() Zoi, 35200.000000
Žak
Zaposlenik() Žak, 52800.000000
stuff:
Šef,10000.000000
Ana, 17600.000000
Zoi, 35200.000000
Žak, 52800.000000
 DŽo
~Zaposlenik() DŽo
Pla?a: 10000.000000
 ~Menadžer() Šef,10000.000000
~Zaposlenik() Šef
~Radnik() Ana, 17600.000000
~Zaposlenik() Ana
~Radnik() Zoi, 35200.000000
~Zaposlenik() Zoi
~Radnik() Žak, 52800.000000
~Zaposlenik() Žak