DOI: 10.19279/TVZ.PD.2015-3-1-04

OSVRT NA C++ 11 STANDARD – SEMANTIKA PRIJENOSA I PAMETNI POKAZIVAČI

C++11 STANDARD REVIEW – MOVE SEMANTICS AND SMART POINTERS

Željko Kovačević, Miroslav Slamić

Tehničko veleučilište u Zagrebu

Sažetak

Programski jezik C++ neprestano se razvija te je do sada objavljeno nekoliko C++ standarda. Međutim, zadnjim C++ 11 standardom ovaj programski jezik dobio je mnogo novih mogućnosti. Ispravljeni su određeni nedostatci samog programskog jezika te su dodane mnoge nove značajke koje olakšavaju način na koji se piše i dodatno optimizira programski kod. Popis svih značajki koje dolaze s C++ 11 standardom poprilično je velik. Stoga su u ovom radu opisane neke od najvažnijih poput semantike prijenosa i pametnih pokazivača. Upotrebom ovih novih značajki C++ programi postaju još brži i sigurniji za pisanje pa je zato i njihovo razumijevanje od ključne važnosti.

***Ključne riječi**: programiranje, c++, c++11, standard, pokazivači, semantika prijenosa.*

Abstract

C++ programming language is continuously being developed. Up to this date several C++ standards have been released. However, C++ 11 Standard has provided this programming language with many new possibilities. Certain disadvantages of the programming language have been corrected. Also, many new properties have been added, which makes it easier to write and additionally optimise the programming code. The list of new properties which come with C++ 11 Standard is quite long. Therefore, this paper will describe some of the most important properties such as move sematics and smart pointers. Using these new features C++ applications are becoming faster and safer for writing. Hence, their understandind is of key value.

***Keywords:** programming, c++, c++11, standard, pointers, move semantics*

1. Uvod

1. Introduction

Kada je riječ o pisanju visoko optimiziranog programskog koda gdje nam je bitna brzina izvođenja, programski jezik C++ je jedno od najboljih rješenja . Iako su u praksi najčešće sporiji, programski jezici novijih generacija su također imali određene značajke kojima su se isticali u usporedbi s C++om. Jedna od tih značajki je, primjerice, sakupljač smeća (eng. *garbage collector*) koji je automatski uništavao dinamički alocirane objekte nakon što isti nisu više bili u upotrebi (dosegu). Tu su također i lambda izrazi (anonimne funkcije), delegati itd. Stoga, unatoč svojoj brzini C++ je na neki način trebao ostati u koraku s najnovijim tehnikama programiranja koje se koriste u novijim objektno orijentiranim programskim jezicima.

Još 1998. g. pojavljuje se prvi C++ standard (ISO/IEC 14882:1998) popularno zvan C++98. Programskom jeziku C++ ovim su standardom dodane prve ozbiljnije značajke i mogućnosti. Tako je primjerice sada bilo moguće koristiti `dynamic_cast` pretvorbu, `typeid` operator, kontejnere, algoritme, iteratore, funkcijske objekte itd. Također, jedna od važnijih značajki koju je uveo C++ 98 bila je prva implementacija pametnog pokazivača (`auto_ptr`) koji će naknadno daljnjim revizijama standarda biti zamijenjen pametnim pokazivačima `unique_ptr` i `shared_ptr`.

C++ standard kasnije je doživio još nekoliko manjih revizija. Prva od njih je bila 2003. godine pod nazivom C++03, a njome se ispravilo mnoštvo prijavljenih problema s prethodno objavljenim C++98 standardom. Kasnije 2005. C++ komisija za razvoj objavila je prvi tehnički izvještaj (TR1) detaljno opisujući razne nove

planirane značajke. Novi standard neformalno je nazvan C++0X te je konačno objavljen tek sredinom 2011. g. Nakon toga je preimenovan u C++11. [1]

Tek od tada službeno C++ podržava mnoštvo novih tehnika i pristupa koji se mogu primjenjivati pri pisanju C++ aplikacija. Jedna od najvažnijih je semantika prijenosa kojom se omogućuje dodatna optimizacija pri razvijanju aplikacija. Naime, sada C++ aplikacija može prepoznati da li se radi s privremenim ili stalnim objektima i na osnovu toga dodatno ubrzati izvršavanje aplikacija. [2]

Na području sigurnosti su također mnogi noviteti, a najvažniji od njih su pametni pokazivači. Pomoću njih C++ programeri više ne moraju ručno dealocirati zauzetu memoriju već se ona automatski briše nakon što više nije potrebna. [3]

2. Prednosti semantike prijenosa u C++ aplikacijama

2. Advantages of move semantics in C++ applications

Programski jezik C++ oduvijek je glasio kao jedan od najbržih, omogućavajući programerima pisanje programskog koda s visokim stupnjem optimizacije. Objavom C++11 standarda ovaj programski jezik postao je čak još brži i optimiziraniji. Najveći razlog tome je upravo korištenje semantike prijenosa (eng. *move semantics*) kojom se eliminiraju nepotrebna kopiranja objekata tamo gdje je to moguće, a time i smanjuje opterećenje procesora pri radu aplikacije.

Za potpuno razumijevanje semantike prijenosa potrebno je razlikovati dva tipa vrijednosti, tj. `lvalue` i `rvalue`. Prvi tip predstavlja izraz koji se nalazi s lijeve strane operatora pridruživanja te ga je moguće adresirati. Zato je `lvalue` najčešće nekakav objekt ili funkcija.

```
int x = 3 + 4;
```

U ovoj deklaraciji `x` je `lvalue` vrijednost jer je riječ o objektu koji postoji i nakon evaluacije izraza s desne strane te ga je moguće adresirati. S druge strane `3 + 4` predstavlja `rvalue` vrijednost i to samo zato jer je riječ o privremenoj vrijednosti koja više neće postojati nakon evaluacije izraza. Zapravo, rvalue vrijednosti privremeni su objekti koji se upotrebom C++11 standarda mogu detektirati te dodatno iskoristiti prilikom upotrebe semantike prijenosa.

Kako C++11 standardom možemo detektirati je li funkciji predana `lvalue` ili `rvalue` vrijednost tako možemo izvršiti i dodatne optimizacije programskog koda. Primjerice,

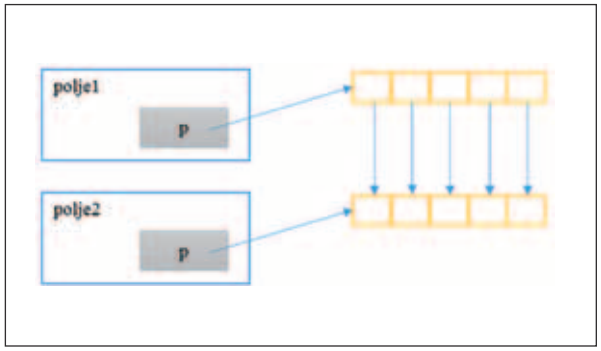
```
class MojeIntPolje{
      public:
            int* p;
            int size;
            MojeIntPolje(int n) :
size(n){
                  p = new int[n];
            }
            // Kopirni konstruktor
(duboko kopiranje)
            MojeIntPolje(const
MojeIntPolje& X){
                  p = new int[X.
size];
                  for(int i = 0; i  <
X.size; i++)
                        p[i] =
X.p[i];
                  size = X.size;
            }
            ~MojeIntPolje(){
                  delete[] p;
            }
};
```

Ova klasa predstavlja polje podataka tipa `int`. U sebi sadrži pokazivač na niz koje se alocira unutar konstruktora te uništava u destruktoru klase. Deklarirajmo sljedeće objekte ove klase:

```
MojeIntPolje polje1(10); //
Konstruktor!

MojeIntPolje polje2 = polje1; //
Kopirni konstruktor!

MojeIntPolje
polje3(MojeIntPolje(10)); //
konstruktor (i kopirni konstruktor?)
```

U prve dvije linije koda jasno je što se događa, a problematična je tek zadnja linija čiji rezultat ovisi o prevoditelju. Ono što bismo očekivali jest da se zbog privremenog objekta `MojeIntPolje(10)` prvo pokrene konstruktor s parametrima, a nakon toga kopirni konstruktor koji bi svojstva tog privremenog objekta kopirao u objekt `polje3`. Ipak, ukoliko vaš prevoditelj koristi RVO (eng. *Return Value Optimization*) onda bi se zadnja linija koda interpretirala kao `MojeIntPolje polje3(10)`. Ovime prevoditelj izbjegava kopirni konstruktor i izravno stvara objekt na osnovu parametara koji su proslijeđeni privremenom objektu.

RVO optimizacija nije karakteristika svih prevoditelja i zato s takvim kodom treba biti na oprezu kako aplikacija ne bi izgubila puno na performansama tokom izvođenja.



Slika 1 *Semantika kopiranja*
Figure 1 *Copy semantics*

Kao što je vidljivo iz slike korištenjem semantike kopiranja (kopirni konstruktor) oba objekta dobila su nove memorijske lokacije za svoje pokazivače, a nakon toga još je potrebno izvršiti kopiranje podataka iz jednog dinamički alociranog niza u drugi. Sve ovo nužno je potrebno ukoliko je riječ o kopiranju iz jednog već postojećeg objekta (`lvalue` vrijednosti) u drugi objekt. Ono što se dodatno optimiziralo kroz C++11 kopiranje je `rvalue` vrijednosti, tj. privremenog objekta u novi objekt. U tu svrhu sada se može koristi semantika prijenosa.

U prethodnom primjeru demonstrirali smo oba poziva:

```
MojeIntPolje polje2 = polje1;

MojeIntPolje
polje3(MojeIntPolje(10));
```

U prvom slučaju `polje2` objekt poprima vrijednost objekta `polje1`. Međutim, `polje1` je `lvalue` vrijednost jer ju je moguće adresirati te će postojati i dalje nakon izvršenja tog izraza. Zbog toga je nužno da se izvrši kopirni konstruktor s dubokim kopiranjem. S druge strane, u drugoj naredbi objektu `polje3` dodjeljujemo `rvalue` vrijednost tj. privremeni objekt.

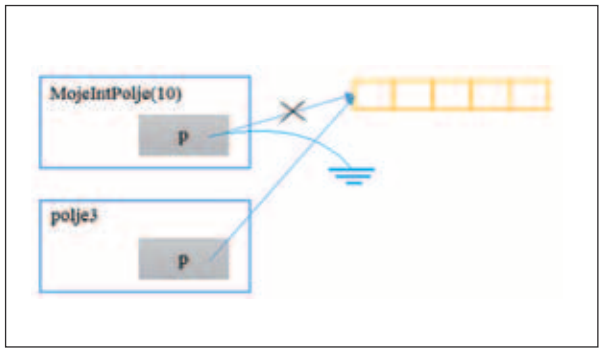
Problem s drugom naredbom jest da je privremeni objekt `MojeIntPolje(10)` već dinamički alocirao niz od 10 elemenata te će ga uništiti odmah nakon izvršenja kopirnog konstruktora objekta `polje3`. Pitanje je onda zašto uopće raditi kopiju privremenog objekta i time gubiti procesorsko vrijeme ako već postojeći niz privremenog objekta njemu ionako više neće trebati? Stoga, umjesto da radimo kopiju privremenog objekta možemo prenijeti njegova postojeća svojstva (dinamički alocirani niz) u novi objekt. Upravo to je semantika prijenosa.

Prilikom realizacije semantike prijenosa koriste se prijenosni konstruktor (eng. *move constructor*) i operator pridruživanja sa semantikom prijenosa. Kao parametre im predajemo `rvalue` reference, a one umjesto jednog znaka '`&`' sadrže dva znaka '`&&`'. Primjerice,

```
// Konstruktor prijenosa s rvalue
referencom

MojeIntPolje(MojeIntPolje&&
privremeni){

      p = privremeni.p; // Preusmjeri
pokazivač

      privremeni.p = NULL; //
privremeni objekt više nije vlasnik
pokazivača

      size = privremeni.size;

}
```

Ovako bi izgledao prijenosni konstruktor za gore opisanu klasu. Kao parametar prima `rvalue` referencu i umjesto rezerviranja nove memorijske lokacije i kopiranja (kopirni konstruktor) jednostavno preuzima pokazivač iz privremenog objekta.

Što se točno događa u kodu vidljivo je iz slike. Objekt `polje3` pomoću prijenosnog konstruktora



Slika 2 *Semantika prijenosa*
Figure 2 *Move semantics*

preuzima pokazivač iz privremenog objekta, a zatim se pokazivač privremenog objekta postavlja na vrijednost NULL. To je potrebno iz razloga što će se prilikom uništenja privremenog objekta pokrenuti njegov destruktor koji bi taj pokazivač dealocirao, a to ne smijemo dopustiti budući da tu memorijsku lokaciju (niz) sada koristi novi objekt.

Još jednom treba napomenuti da zbog RVO optimizacije vaš prevoditelj možda uopće neće pokrenuti prijenosni ili kopirni konstruktor, ali ga na to možete eksplicitno prisiliti upotrebom funkcije `move`.

```
MojeIntPolje polje3(move(MojeIntPol
je(10)));
```

Tablica 1 *Vremensko trajanje kreiranja i kopiranja privremenih objekata*

Table 1 *Time required for creating and copying temporary objects*

Broj objekata	bez RVO (ms)	RVO (ms)	Konstr. prijenosa (ms)
100.000	31	16	18
1.000.000	261	103	110
2.000.000	493	197	209
5.000.000	1145	415	430
10.000.000	2296	847	899
20.000.000	4624	1650	1713

Ovom tablicom pokazano je koliko vremena treba za kreiranje i kopiranje određenog broja privremenih objekata ukoliko prevoditelj koristi ili ne koristi RVO optimizaciju te ukoliko se koristi konstruktor prijenosa. Naravno, u ovisnosti o računalu i prevoditelju rezultati testiranja mogu varirati, a za ovaj test korišten je prevoditelj Clang (64 bit) te C++ Builder XE6.

Ukoliko prevoditelj ne koristi RVO optimizaciju to znači da se prvo pokreće konstruktor privremenog objekta a zatim i kopirni konstruktor koji će taj privremeni objekt kopirati u destinaciju upotrebom dubokog kopiranja. Zbog toga ovaj test očekivano najduže traje.

Korištenjem RVO optimizacije prevoditelj će preskočiti pozivanje kopirnog konstruktora te time uštedjeti vrijeme. Ovo je svakako najbrža varijanta jer umjesto da kreiramo privremeni objekt kojeg zatim treba kopirati u destinaciju mi odmah u samoj destinaciji kreiramo željeni objekt.

U zadnjem slučaju se koristi prijenosni konstruktor koji također izbjegava duboko kopiranje. Njegovo izvršavanje traje tek nešto duže nego li izvršavanje u slučaju korištenja RVO optimizacije. Razlog tome je što nakon pozivanja konstruktora privremenog objekta poziva se konstruktor prijenosa. Međutim, konstruktor prijenosa radi tek plitke kopije podataka, što traje mnogo manje vremena nego li korištenje kopirnog konstruktora koji radi duboko kopiranje.

Iz ove analize se može zaključiti da je korištenje RVO optimizacije još uvijek najbolje rješenje. No kako to još uvijek nije odlika svih prevoditelja alternativno se može koristiti prijenosni konstruktor koji po performansama daje gotovo identične rezultate.

Kao i u slučaju kopirnog konstruktora i operatora pridruživanja s dubokim kopiranjem, tako isto možemo definirati i operator pridruživanja sa semantikom prijenosa. Za gornju klasu on bi izgledao na sljedeći način.

```
MojeIntPolje& operator =
(MojeIntPolje&& privremeni){
      if (this != &privremeni){
            delete[] p;
            p = privremeni.p;
            privremeni.p = NULL;
            size = privremeni.size;
      }
      return *this;
}
```

Ukoliko pretpostavimo da naša klasa ima podrazumijevani konstruktor, primjer upotrebe bio bi sljedeći.

```
MojeIntPolje polje4;

polje4 = MojeIntPolje(10); //
operator pridruživanja sa semantikom
prijenosa
```

Također je i u ovom slučaju moguće koristiti funkciju `move` ukoliko prevoditelj bude koristio RVO optimizaciju.

Upotrebom semantike prijenosa C++ postaje još brži programski jezik. Ova značajka je također integrirana i u kontejnere koji za svoje interne operacije koriste privremene objekte (`rvalue` vrijednosti).

3. Korištenje pametnih pokazivača

3. Using smart pointers

Dugo je vremena sakupljač smeća (eng. *garbage collector*) bio jedna od prednosti programskih jezika poput C# i Jave. Svojevremeno, i C++ je razvijao svoju alternativu u smislu pametnih pokazivača. Oni bi nakon izlaska iz dosega automatski dealocirali zauzetu memoriju, čak i u slučajevima kada bi se dogodile iznimke. Sada programer više ne mora misliti o dealokaciji dinamički alociranih objekata što uvelike olakšava pisanje aplikacija.

Da bismo razumjeli pametne pokazivače prvo je potrebno shvatiti koncept vlasništva jer ovisno o tome C++11 nudi više različitih tipova pametnih pokazivača.

```
int *p = new int;
…
delete p;
```

Za dinamičku alokaciju memorije do sada smo najčešće koristili ovakav oblik koda pomoću operatora `new` i `delete`. Ovdje programer mora voditi brigu o dealokaciji memorije novog objekta te se stoga može reći da je upravo programer njegov vlasnik. Ali ako smo za dinamičku alokaciju objekta koristili pametni pokazivač, tada programer više nije vlasnik tog novog objekta jer brigu o njegovoj dealokaciji preuzima upravo taj pametni pokazivač.

```
#include <memory>

std::auto_ptr<int> p(new int); // p
postaje vlasnik novog objekta
```

Prvi pokušaj realizacije bio je pomoću `auto_ptr` pametnog pokazivača. Svaki pametni pokazivač tog tipa je bio vlasnik jednog dinamički alociranog objekta o kojemu je vodio brigu. Štoviše, nije se moglo dogoditi da dva pametna pokazivača imaju vlasništvo nad istim objektom jer se tada ne bi znalo tko je zadužen za njegovu dealokaciju. Primjerice,

```
auto_ptr<int> p(new int);

auto_ptr<int> p2 = p; // p2 sada
postaje vlasnik objekta!

*p = 1; // nedefinirano ponašanje!
```

Pokušate li prevesti ovaj dio programskog koda vaš prevoditelj neće javiti grešku, dok će neki prevoditelji javiti tek upozorenje da je korištenje `auto_ptr` pametnog pokazivača zastarjelo. O čemu se zapravo radi?

Prvi pametni pokazivač `p` dinamički je alocirao objekt tipa `int` te je on njegov isključivi vlasnik. Već u drugoj liniji koda kreira se drugi pametni pokazivač `p2` koji koristi semantiku kopiranja kako bi preuzeo vlasništvo prethodno alociranog objekta. Čim je `p2` preuzeo vlasništvo nad tim objektom, zadnja linija koda predstavlja nedefinirano ponašanje jer `p` nije više vlasnik nikakvog objekta.

Ovdje vidimo koliko je ovaj kod opasan i nesiguran, a pogotovo pri radu s kontejnerima koji bi interno izvršavali operacije kopiranja elemenata radi sortiranja, umetanja itd. ukoliko bi elementi bili upravo `auto_ptr` pametni pokazivači. Izvršavanjem tih operacija pomoću semantike kopiranja kontejneri bi implicitno mijenjali i vlasnike objekata čime bi kod postao potpuno nepredvidiv i nesiguran.

Sve ovo događa se jer je pametni pokazivač `auto_ptr` pisan u vrijeme kada C++ nije imao semantiku prijenosa te je koristio isključivo semantiku kopiranja za prijenos vlasništva. Danas C++11 donosi i semantiku prijenosa te novi pametni pokazivač koji u potpunosti zamjenjuje `auto_ptr`. Riječ je o unique_ptr pametnom pokazivaču. Nakon njegovog uvođenja korištenje

auto_ptr pametnog pokazivača smatra se zastarjelim te se više ne preporučuje njegova upotreba.

```
unique_ptr<int> p(new int);

unique_ptr<int> p2 = p; // greška prevoditelja!
```

Iz ovog primjera vidljivo je kako unique_ptr uopće ne dopušta korištenje semantike kopiranja za prijenos vlasništva, već da bi to bilo moguće mora se koristiti semantika prijenosa pomoću funkcije move.

```
unique_ptr<int> p2 = move(p); // ispravan prijenos vlasništva!
```

Stoga, korištenje unique_ptr pametnog pokazivača puno je sigurnije jer će prevoditelj pri ovakvom pokušaju javiti grešku te neće dopustiti nedefinirana ponašanja kao u slučaju korištenja auto_ptr pametnog pokazivača.

Ovaj pametni pokazivač ima i druge pogodnosti, poput dopuštanja dinamičke alokacije niza objekata. To prethodno nije bilo moguće upotrebom auto_ptr pametnog pokazivača jer bi on implicitno uvijek pozivao operator delete umjesto delete[] kada bi to bilo potrebno.

```
unique_ptr<int[]> niz(new int[10]);
```

Korištenjem unique_ptr-a sigurni smo da će se pri dealokaciji osloboditi cijeli niz, tj. pozvati delete[]. Naravno, ukoliko je alociran tek jedan objekt automatski će biti pozvan samo delete.

I sama dealokacija može se dodatno specificirati korištenjem deleter izraza. Naime, unique_ptr pametni pokazivači podržavaju mogućnost da programer sam definira funkcijski objekt koji bi bio zadužen za dealokaciju objekta. Primjerice,

```
class MojDeleteObjekt{
public:
	template <class T>
	void operator()(T* p) {
		delete p;
		cout << "Objekt je dealociran!\n";
	}
};
…
unique_ptr<int, MojDeleteObjekt> niz(new int);
```

ili

```
MojDeleteObjekt del;

unique_ptr<int, MojDeleteObjekt&> niz(new int, del);
```

U oba ova slučaja koristimo vlastite funkcijske objekte koji će osim dealokacije objekta moći obaviti i neke dodatne operacije ukoliko je to potrebno.

Također treba napomenuti da unatoč pogodnostima pametnih pokazivača većina ugrađenih funkcija koristi upravo obične pokazivače. Stoga se često javlja potreba da se iz pametnog pokazivača dohvati njegov interni (obični) pokazivač na alocirani objekt.

```
void inkrement(int* p){
	(*p)++;
}
```

Pitanje je kako ovoj funkciji predati pametni pokazivač, tj. njegov interni pokazivač koji pokazuje na alocirani objekt. To možemo postići na sljedeći način:

```
unique_ptr<int> n(new int(1));

inkrement(n.get()); // get() vraća interni pokazivač na objekt

cout << *n; // 2  (dereferenciranje pametnog pokazivača)
```

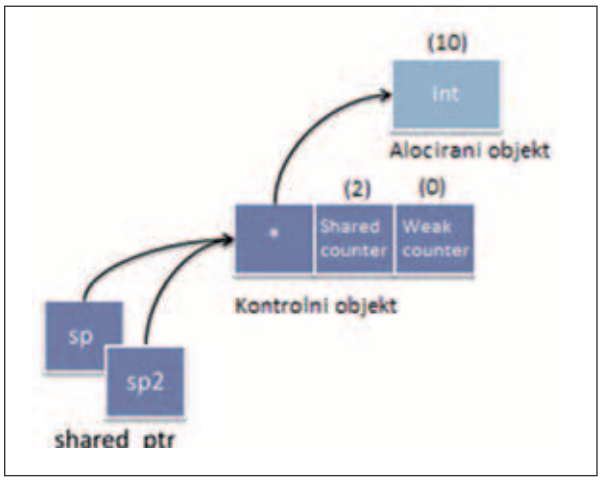
Metodom get() iz pametnog pokazivača n dohvaćamo njegov interni (obični) pokazivač na alocirani objekt. Taj interni pokazivač onda prosljeđujemo funkciji inkrement. Ipak, u ovakvim slučajevima treba biti na oprezu ukoliko nismo sigurni što dotična funkcija radi. Primjerice, takva funkcija može dealocirati memoriju na koju pokazuje pokazivač te time učiniti pametni pokazivač beskorisnim.

Iako se unique_ptr danas smatra najpogodnijim za korištenje postoji i druga alternativa. Primjerice, unique_ptr garantira da postoji samo jedan vlasnik alociranog objekta. Dozvoljeno je tek prenositi to vlasništvo na drugi unique_ptr pokazivač upotrebom semantike prijenosa. No postoje i situacije gdje nas to u

potpunosti ne zadovoljava, odnosno upravo kada trebamo mogućnost da jedan objekt ima više vlasnika. Zbog toga C++11 nudi i `shared_ptr` pametni pokazivač. [4]

```
shared_ptr<int> sp(new int(10));

shared_ptr<int> sp2 = sp; // sp2
postaje još jedan vlasnik objekta!

cout << *sp2; // 10
```

Alocirani objekt sada ima dva vlasnika. Ovakva situacija ne bi bila dopuštena upotrebom `unique_ptr` pokazivača jer bi prevoditelj zahtijevao prijenos vlasništva funkcijom `move`.



Slika 3 *Približni memorijski raspored za prethodni primjer*
Figure 3 *Approximate memory layout for the previous example*

Proces započinje dinamičkom alokacijom željenog objekta nakon čega se pokreće konstruktor `shared_ptr` objekta sp. Njime se stvara kontrolni objekt koji pokazuje na alocirani objekt. Kontrolni objekt interno prati koliko drugih `shared_ptr` i `weak_ptr` pokazivača koristi alocirani objekt te sukladno tome povećava ili smanjuje njihove brojače. [3]

Tako se kreiranjem pametnog pokazivača sp povećava brojač `shared_ptr` objekata (`shared counter`) za 1. Nakon kreiranja drugog pametnog pokazivača `sp2` koji ima vlasništvo nad istim alociranim objektom taj brojač se opet povećava za 1 te trenutno iznosi 2. Ali kada jedan od tih pametnih pokazivača izađe iz dosega tada se i njihov brojač umanjuje za 1. Konačno, kada brojač `shared counter` poprimi vrijednost 0 (svi `shared_ptr`-i su izašli iz dosega) događa se uništenje alociranog objekta (dealokacija).

Treba napomenuti da uništenje alociranog objekta ne znači nužno i uništenje kontrolnog objekta. On će i dalje postojati sve dok brojač `weak counter` također ne poprimi vrijednost 0. Bez obzira na uništenje alociranog objekta, kontrolni objekt treba pružiti informaciju `weak_ptr` pokazivačima o postojanosti alociranog objekta. Zato, sve dok postoji barem jedan `weak_ptr` pokazivač koji je bio vezan za jedan od prethodnih shared_ptr pokazivača kontrolni objekt će postojati. O `weak_ptr` pokazivačima bit će riječ u nastavku.

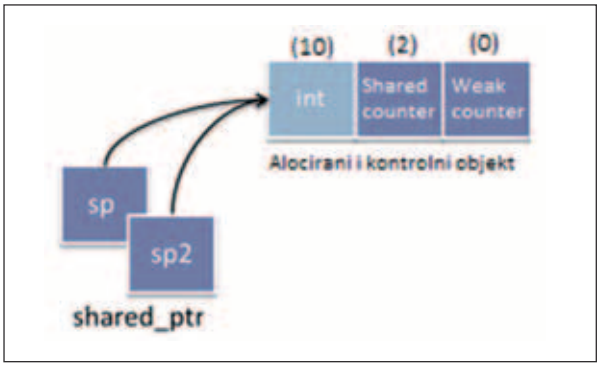
U prethodnom primjeru vidjeli smo što se događa nakon izvršenja sljedeće linije koda:

```
shared_ptr<int> sp(new int(10)); //
dvostruka alokacija!
```

Dogodila se dvostruka alokacija memorije. Prva alokacija je pri pozivanju operatora `new`, a druga pri pozivanju `shared_ptr` konstruktora koji će kreirati kontrolni objekt (pogledati sliku 3). Da bismo to izbjegli možemo koristiti `make_shared` funkciju.

```
shared_ptr<int> sp(make_
shared<int>(10)); // samo jedna
alokacija!
```

Funkcija `make_shared` samo će u jednoj alokaciji kreirati traženi objekt i njegov kontrolni objekt. U memoriji bi to izgledalo na sljedeći način:



Slika 4 *Korištenje make_shared funkcije*
Figure 4 *Using make_shared function*

Upravo iz razloga što dinamičke alokacije memorije najduže traju, preporuča se korištenje `make_shared` funkcije kao dodatnog načina optimizacije.

Tablicom su prikazani rezultati testiranja pri kreiranju određenog broja objekata upotrebom

Tablica 2 *Rezultati testiranja pri kreiranju shared_ptr objekata*
Table 2 *Testing results upon creating shared_ptr objects*

Broj objekata	shared_ptr (ms)	make_shared (ms)
100.000	46	31
1.000.000	422	390
2.000.000	843	702
5.000.000	2012	1701
10.000.000	4068	3432
20.000.000	7921	6443

`shared_ptr` predloška i `make_shared` funkcije. Rezultati mogu varirati u ovisnosti o konfiguraciji na kojoj se testiranje izvodi, no u ovom slučaju mogu se primijetiti i preko 20% bolje performanse upotrebom `make_shared` metode. Jasno, razlog boljim performansama metode `make_shared` je samo jedna memorijska alokacija. Za testiranje je korišten Clang (64 bit) prevoditelj te C++ Builder XE6 razvojno okruženje.

Jedini ozbiljniji problem koji se može javiti pri korištenju `shared_ptr` pametnih pokazivača cirkularne su ovisnosti. Primjerice, dva ili više `shared_ptr` pokazivača mogu jedan drugog „održavati na životu“ te time u konačnici uzrokovati curenje memorije (eng. *memory leak*).

```
class Osoba{
public:
	string ime;
	shared_ptr<Osoba> prijatelj;
	// konstruktor
Osoba(string _ime) : ime(_ime){
		cout << ime << " je
stvoren!\n";
	}
	// destruktor
	~Osoba(){
		cout << ime << " je
unisten!\n";
	}
	// dodaj novog prijatelja
(shared_ptr)
	void dodajPrijatelja(shared_
ptr<Osoba> _prijatelj){
		prijatelj = _prijatelj;
	}
};
```

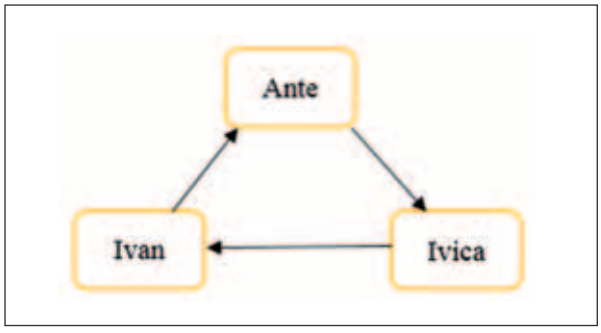
Za demonstraciju cirkularne ovisnosti kreirat ćemo 3 osobe pomoću `shared_ptr` pametnih pokazivača.

```
shared_ptr<Osoba> Ante(make_
shared<Osoba>("Ante"));

shared_ptr<Osoba> Ivica(make_
shared<Osoba>("Ivica"));

shared_ptr<Osoba> Ivan(make_
shared<Osoba>("Ivan"));
```

Svaka od kreiranih osoba može biti prijatelj s drugom osobom. Primjerice,

```
// cirkularne reference

Ante->dodajPrijatelja(Ivica); // Ante
-> Ivica

Ivica->dodajPrijatelja(Ivan); //
Ivica -> Ivan

Ivan->dodajPrijatelja(Ante); // Ivan
-> Ante
```

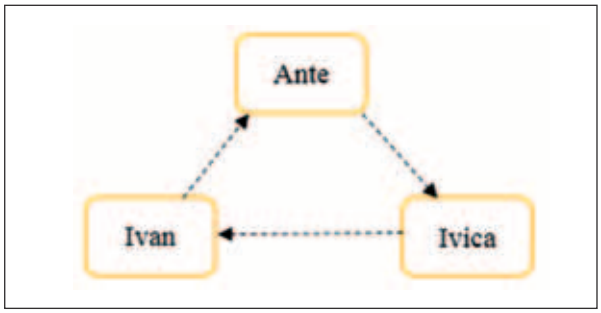
Izvršavanjem ovog programskog koda nikada se neće pokrenuti niti jedan od destruktora za 3 alocirana objekta. To je upravo zato jer jedan drugog „održavaju na životu“ zbog međusobne čvrste veze `shared_ptr` pokazivačima.



Slika 5 *Cirkularna ovisnost pomoću shared_ptr pokazivača*
Figure 5 *Circular dependency using shared_ptr pointers*

Zbog ovakvih situacija uvedeni su `weak_ptr` pokazivači. Njima se definira „slaba veza“ između `shared_ptr` pokazivača, odnosno prekida se cirkularna ovisnost. Za `weak_ptr` pokazivače možemo reći da tek promatraju objekt ne utječući na njegov životni vijek. Sukladno tome, ukoliko podatkovni član

`shared_ptr<Osoba>` prijatelj deklariramo kao `weak_ptr<Osoba>` prijatelj imat ćemo sljedeću situaciju:



Slika 6 *Veze pomoću weak_ptr pokazivača*
Figure 6 *Connections with weak_ptr pointers*

Nakon što sada izvršimo gornji programski kod uredno će se izvršiti destruktori za sve 3 kreirane osobe.

S obzirom da `weak_ptr` pokazivačima ne definiramo čvrstu vezu među `shared_ptr` objektima nikada ne znamo je li objekt na koji `weak_ptr` pokazuje još uvijek živ. Ipak, to možemo provjeriti metodom `lock()`. Ova metoda stvara privremeni `shared_ptr` objekt. Ukoliko objekt na koji `weak_ptr` pokazuje nije živ tada je privremeni `shared_ptr` objekt prazan.

```
void Promatraj(std::weak_ptr<int>
weak){
    std::shared_ptr<int> pom(weak.
lock());
    if (pom) {
        std::cout << "Objekt postoji.
Vrijednost je " << *pom << "\n";
    } else {
        std::cout << "Objekt vise ne
postoji!\n";
    }
}
```

Funkcija `Promatraj` prima `weak_ptr` objekt, a zatim metodom `lock()` ispituje je li objekt na koji on pokazuje još uvijek živ.

```
int main(){
    std::weak_ptr<int> weak;
    {
        std::shared_ptr<int>
shared(new int(10));
        weak = shared; // stvara se
"slaba veza"
        Promatraj(weak); // shared
objekt postoji!
    }
    Promatraj(weak); // shared objekt
više ne postoji!
}
```

Treba primijetiti da `shared_ptr` objekt `shared` postoji u zasebnom bloku. Ukoliko ga u tom bloku promatramo pomoću `weak_ptr` pokazivača neće biti problema. Tek nakon izlaska iz tog bloka objekt `shared` izlazi iz dosega te je automatski dealociran, a to se detektira zadnjim pozivom funkcije `Promatraj`.

4. Zaključak

4. Conclusion

C++11 standard donosi gotovo revolucionarne pomake prema naprijed jer C++ sada konačno ima sve značajke koje su mu još nedavno prigovarane kao glavni nedostatci. Ovaj rad opisuje tek neke od tih značajki dok je puni popis mnogo veći. C++ programerima donose se poboljšanja na svim poljima, bilo da je riječ o poboljšanju performansi, sigurnosti izvršavanja koda ili stilu pisanja. Jedini problem u ovom trenutku jest što svi prevoditelji još uvijek ne podržavaju C++11 ili ga podržavaju tek djelomično. Proći će još neko vrijeme dok integracija ne bude potpuna, a tada će se postupno i standard početi sve više primjenjivati.

5. Reference

5. References

[1] A. S., »History of C++,« [Mrežno]. Available: http://www.cplusplus.com/info/history/. [Pokušaj pristupa 13 7 2014].

[2] A. Allain, »Move semantics and rvalue references in C++11,« [Mrežno]. Available: http://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html. [Pokušaj pristupa 14 7 2014].

[3] H. Sutter, »Smart Pointers,« 29 5 2013. [Mrežno]. Available: http://herbsutter.com/2013/05/29/gotw-89-solution-smart-pointers/. [Pokušaj pristupa 16 7 2014].

[4] D. Kieras, »Using C++11's Smart Pointers,« Michigan, 2013.

AUTORI · AUTHORS

Željko Kovačević, struč. spec. ing. techn. inf
- nepromjenjena biografija nalazi se u časopisu Polytechnic & Design Vol. 2, No. 1, 2014.

Korespondencija:
zeljko.kovacevic@tvz.hr

Miroslav Slamić is currently professor-lecturer at Polytechnic of Zagreb. Courses that he teaches are related to algorithms and data structure, and programming languages and paradigms. He obtained his PhD in Aviation, Rocket and Space technology Science from Faculty of Electrical Engineering and Computing Zagreb. His academic research interest include interactive real-time simulations, virtual reality and applications in staff training, diagnostics and therapy. He also worked on management of massive multimedia contents. Special scientific interest and research projects he had in missile guidance systems and theory of differential games. He has more than 25 professional, academic and scientific papers published in these areas.