Lekcija 08

Uvod u C++ klase i objekte

Miljan Milošević

UVOD U C++ KLASE I OBJEKTE

	01	02	03	04
Uvod	**Osnovi o klasama i objektima**	**Funkcije članice i podaci**	**Konstruktor i destruktor**	**Pokazivač this**
	➢ *Osnovni pojmovi* ➢ *Obične promenljive i objekti - poređenje* ➢ *Definicija klasa* ➢ *Deklaracija objekata* ➢ *Pristup podacima članovima* ➢ *Detaljan uvid u klase i objekte*	➢ *Definicija i upotreba funkcija članica* ➢ *Primer definicije i upotrebe funkcija članica klase* ➢ *Uvod u funkcije pristupa članovima klase - geter i seter metode* ➢ *Uvod u funkcije pristupa članovima klase - geter i seter metode* ➢ *Linijske (inline) funkcije*	➢ *Konstruktori* ➢ *Destruktor* ➢ *Konstruktor kopiranja*	➢ *Osnovi o pokazivaču this* ➢ *Upotreba pokazivača this*

UVOD U C++ KLASE I OBJEKTE

05

Odvojena definicija klase

- *Osnovna razmatranja*
- *Primer odvojene definicije klase*

06

Područje (oblast važenja) klase

- *Oblast važenja klase*
- *Oblast važenja kod definicije klase u odvojenom fajlu*

07

Konstantne funkcije i objekti

- *Konstantni (const) objekti*
- *Konstantne funkcije članice klase*
- *Pravilno korišćenje konstantnih funkcija*
- *Primeri upotrebe konstantnih funkcija*

08

Specifikatori pristupa članovima klase

- *Tipovi specifikatora pristupa*
- *Javni članovi klase - public*
- *Privatni članovi klase - private*
- *Zaštićeni članovi klase - protected*

09

Vežbe

- *Redosled poziva konstruktora i destruktora*
- *Studija slučaja – Kreiranje i upotreba klase Time*

10

Zadaci za samostalan rad

- *Zadaci za samostalno vežbanje*
- *Zadaci za samostalno vežbanje - Dodatak*

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

U okviru ove lekcije studenti se upoznaju sa osnovnim pojmovima objektno orijentisanog principa programskog jezika C++:

- Osnovi o klasama i objektima
- Funkcije članice i podaci
- Konstruktor i destruktor
- Pokazivač this
- Odvojena definicija klase
- Područje (oblast važenja) klase
- Konstantne funkcije i objekti
- Specifikatori pristupa članovima klase

Osnovna svrha C++ programiranja je da se doda objektna orijentacija C programskom jeziku. Klase (class) su centralna odlika C++-a koja upravo podržava objektno orijentisano programiranje. Po sintaksi se deklaracija klase može najbliže povezati sa načinom deklarisanja struktura podataka. U skladu sa principima OOP klase za razliku od struktura, osim podataka sadrže i funkcije članice (member functions). Same strukture u sintaksi jezika C++ mogu sadržavati funkcije, ali ono što ih razlikuje od klasa je da ne podržavaju osnovne osobine koje se zahtevaju od klasa i objekata i te osobine predstavljaju suštinu OOP. To su: nasleđivanje (inheritance), apstrakcija (abstraction) i polimorfizam (polymorphism).

Osnovi o klasama i objektima

definicija klase, definicija objekta, polja klase, pristup podacima

- *Osnovni pojmovi*
- *Obične promenljive i objekti - poređenje*
- *Definicija klasa*
- *Deklaracija objekata*
- *Pristup podacima članovima*
- *Detaljan uvid u klase i objekte*

01

OSNOVNI POJMOVI

Osnovna svrha C++ programiranja je da se doda objektna orijentacija C programskom jeziku. Klase su centralna odlika C++-a koja upravo podržava objektno orijentisano programiranje

Osnovna svrha C++ programiranja je da se doda objektna orijentacija C programskom jeziku. Klase (class) su centralna odlika C++-a koja upravo podržava objektno orijentisano programiranje. Njih u kontekstu objektnog jezika C++ treba shvatiti kao specifičan, korisnički definisan tip podataka koji se koristi da opiše realno postojeće objekte. Klase obuhvataju atribute ali i akcije koje služe da simuliraju te realno postojeće objekte. Po sintaksi se deklaracija klase može najbliže povezati sa načinom deklarisanja struktura podataka. U skladu sa principima OOP klase za razliku od struktura, osim podataka sadrže i funkcije članice (member functions). Podaci i metode unutar klase se nazivaju članice klase.

Klase se koriste sa ciljem da se specificira forma objekata. Objekti (object) se mogu shvatiti, oslanjajući se na sintaksu, kao instance (instance) deklarisanih klasa. Konkretna strukturna promenljiva je instanca deklarisane strukture. Isto tako, objekat je instanca deklarisane klase. Ovu vezu klasa i objekata treba imati uvek u vidu jer se u terminologiji često zanemaruje potpuno različit karakter ova dva termina. Klase su dakle šeme objekata, a objekti su konkretni podaci u memoriji računara sa pripadajućim funkcijama definisanim u odgovarajućim klasama.

Bilo bi pogrešno zaključiti da su klase samo nove, dodavanjem funkcija, proširene strukture, a objekti pripadajuće napredne strukturne promenljive. Same strukture u sintaksi jezika C++ mogu sadržavati funkcije, ali ono što ih razlikuje od klasa je da ne podržavaju osnovne osobine koje se zahtevaju od klasa i objekata i te osobine predstavljaju suštinu OOP. To su: nasleđivanje (inheritance), apstrakcija (abstraction) i polimorfizam (polymorphism).

OBIČNE PROMENLJIVE I OBJEKTI - POREĐENJE

U cilju kreiranja objekta potrebno je kreirati „instancu“ tj primerak klase. Ovo se postiže na potpuno isti način kao i kreiranje primeraka tj. „instanci“ običnih tipova podataka

Bilo koji realno postojeći objekat može se definisati preko njegovih atributa i njegovih akcija. Npr. neki pas ima atribute kao što su starost, težina, i boja. Takođe taj pas poseduje akcije koje može da izvršava, npr. da jede, spava i laje. Mehanizam upotrebe klasa u C++-u omogućuje da se kreira virtualni pas u nekom programu. Važno je znati, da definicija klase samo kreira tip podataka koji obuhvata, encapsulates, tj. enkapsulira atribute i akcije. Ali u cilju kreiranja objekta, potrebno je kreirati „instancu“ tj primerak klase. Ovo se postiže na potpuno isti način kao i kreiranje primeraka („instanci“) običnih tipova podataka. Npr. instrukcija:

```
int x;
```

kreira instancu po imenu „x“ , tipa „int“, a instrukcija

```
Pas Jimi;
```

kreira jednu instancu po imenu Jimi, tipa Pas, gde je „Pas“ složeni tip podataka kreiran od strane programera da opiše realno postojeće objekte preko atributa i akcija. Objektima se manipuliše u programu, npr. atributima se inicijalizuju i zadaju vrednosti a metode se pozivaju od strane programa. Npr. instrukcija

```
Jimi.boja = "crna";
```

zadaje vrednost atributu „boja“ primerku tj. instanci Jimi klase Pas. Ovde se koristi, kao što vidimo, „tačka-operator“ odnosno “.“. Vrednosti članova objekata se mogu zadati ili preuzeti pomoću tačka operatora - the dot operator (the member selection operator: ‘.’). Međutim, nije moguće svim članovima objekta pristupiti korišćenjem operatora (.) ali o tome će biti više reči u nastavku.

DEFINICIJA KLASA

Definicija klase počinje ključnom rečju class za kojom sledi ime klase i telo klase oivičeno parom vitičastih zagrada

Pri kreiranju klase se ustvari definiše šablon za neki tip podatka. Definicija klase ne predstavljanje definisanje podatka, ili promenljive, ali definiše značenje klase, tj definiše šta će objekat ove konkretne klase sadržati i koje su to operacije koje će moći da se obavljaju nad objektom.

Definicija klase počinje ključnom rečju class za kojom sledi ime klase i telo klase oivičeno parom vitičastih (velikih) zagrada. Definicija klase se mora završiti sa tačka-zarez znakom. Da bi se iz dela programa van klase omogućio pristup članovima klase, atributima i metodama, koristi se ključna reč public iza koje se stavljaju dve tačke :. U sledećem primeru je definisana nova klasa, odnosno nov tip podakta Box korišćenjem ključne reči class:

```
class Box
{
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};
```

U okviru tela klasa možete primetiti ključnu reč public koja određuje tip pristupa članovima klase koji slede nakon nje. Pristup označen kao public, odnosno javni pristup, znači da je tim članovima klase moguće pristupiti iz bilo kog dela programa u kome je moguće koristiti promenljivu, odnosno objekat te klase. Osim modifikatora public, moguće je koristiti modifikatore private i protected o čemu će biti više reči u toku ove i naredne lekcije. Npr. jedna jednostavna verzija klase Pas može da izgleda ovako

```
class Pas
{
public:
 int starost;
 int tezina;
 string boja;
 void lajati();
};
```

DEKLARACIJA OBJEKATA

Deklaracija objekta se vrši na isti način kako se deklarišu promenljive osnovnih tipova podatka

Klase obezbeđuju šablone za objekte što znači da se objekti kreiraju na osnovu klasa. Deklaracija objekta se vrši na isti način kako se deklarišu promenljive osnovnih tipova podatka. Neki objekat je jedna instanca klase. Pošto se definiše klasa, mogu se zatim u programu kreirati objekti kao instance klase, i može se u programu manipulisati objektima. U nastavku je dat primer deklaracije dva objekta klase Box:

```
Box Box1;  // Declare Box1 of type Box
Box Box2;  // Declare Box2 of type Box
```

Ono što treba znati kod kreiranih objekata da će i Box1 i Box2 imati sopstvene kopije podataka klase dok će funkcije odnosno ponašanja objekata biti zajednička.

Alternativno, neki primerak klase može se kreirati specifikacijom njegovog imena posle velikih zagrada u definiciji klase i stavljanjem znaka tačka-zarez. Višestruki primerci takođe se mogu kreirati na ovaj način, specifikacijom liste imena objekata, uz upotrebu zareza. Primer:

```
class Friend
{
public:
          string surname;
          int phone;
} John, Jane;
```

PRISTUP PODACIMA ČLANOVIMA

Javnim (public) podacima objekata klase može se pristupiti korišćenjem operatora pristupa (.), što nije dozvoljeno kod privatnih (private) i zaštićenih (protected) članova klase

Javnim podacima objekata klase može se pristupiti korišćenjem operatora pristupa (.). Pogledajmo sledeći primer da bi operacija pristupa bila jasnija (koristimo prethodno definisanu klasu Box):

```
#include <iostream>
using namespace std;
class Box {/*ubaciti kod definicije klase Box*/};
int main( )
{
   Box Box1;        // Declare Box1 of type Box
   Box Box2;        // Declare Box2 of type Box
   double volume = 0.0;// Store the volume of a box
here

   // box 1 specification
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.breadth = 7.0;

   // box 2 specification
   Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.breadth = 13.0;
   // volume of box 1
   volume = Box1.height * Box1.length *
Box1.breadth;
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.height * Box2.length *
Box2.breadth;
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

Nakon izvršavanja programa dobija se sledeći rezultat:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Treba naglasiti da se privatnim (private) i zaštićenim (protected) članovima klase ne može direktno pristupiti korišćenjem operatora direktnog pristupa (.) ali o tome će biti više reči u nastavku.

Npr. ako je kreirana klasa Pas, onda se može kreirati objekat „Jimi“, i zadati vrednosti atributima objekta. U nastavku je primer programa gde se definiše klasa i kreira objekat a zatim se manipuliše objektom tj. njegovim atributima:

```
#include <string>
#include<iostream>
using namespace std;
class Pas{/*definicija klase Pas*/};
int main()
{
            Pas Jimi;
            Jimi.starost =3;
            Jimi.tezina = 15;
            Jimi.boja = "crna";
            cout << Jimi.starost << endl;
            cout << Jimi.tezina << endl;
            cout << Jimi.boja << endl;
            return 0;
}
```

DETALJAN UVID U KLASE I OBJEKTE

C++ nudi široku paletu opcija koje olakšavaju kreiranje i korišćenje klasa i objekata

Do sada smo se upoznali sa osnovama u vezi klasa i objekata. Postoji još puno dodatnih koncepata koji se tiču C++ objekata i klasa o kojima će biti više reči u nastavku. Na narednoj slici je lista bitnih podoblasti koje će biti obrađene u okviru ove i naredne lekcije:

Koncept	Opis
Funkcije članice klase	Funkcija članica klase je bilo koja funkcija čija se definicija ili prototip nalazi unutar definicije klase, kao i za ostale elemente klase.
Modifikatori pristupa članovima klase	Član klase može biti definisan kao public, private ili protected. Podrazumevano je da su podaci tipa private.
Konstruktor i destruktor	Konstruktor je specijalna funkcija u klasi koja ima isto ime kao i klasa, i koja se poziva prilikom kreiranja objekta (instance) klase. Destruktor je takođe specijalna funkcija koja se poziva prilikom brisanja objekta.
Konstruktor kopiranja	Konstruktor kopiranja je takav konstruktor koji kreira objekat takav da mu se vrednost inicijalizuje sa vrednošću nekog drugog objekta iste klase, koji je prethodno kreiran.
Linijske(inline) funkcije	Kada naiđe na inline funkciju, kompajler pokušava da zameni poziv funkcije sa kodom definisanim u okviru tela funkcije.
Pokazivač this	Svaki objekat ima specijalan pokazivač **this** koji pokazuje na taj isti objekat.
Statički članovi klase	I podaci klase i funkcije klase mogu biti deklarisani kao statički (static).
Prijateljske funkcije i klase	Prijateljska (**friend**) funkcija je ona kojoj je dozvoljen pun pristup privatnim (private) ili zaštićenim (protected) članovima klase.

Slika-01 Detaljan uvid u klase i objekte. Osnovni pojmovi pri radu sa C++ klasama i objektima

Funkcije članice i podaci

funkcije članice, poziv funkcije, get i set metode, linijske funkcije

- *Definicija i upotreba funkcija članica*
- *Primer definicije i upotrebe funkcija članica klase*
- *Uvod u funkcije pristupa članovima klase - geter i seter metode*
- *Uvod u funkcije pristupa članovima klase - geter i seter metode*

02

DEFINICIJA I UPOTREBA FUNKCIJA ČLANICA

Funkcije članice klase su one funkcije čija se definicija ili prototip nalazi u okviru definicije (tela) klase. One mogu da operišu sa svim objektima klase čiji su član

U prethodnom delu je obrađen skup osnovnih termina i ideja objektno orijentisanog programiranja. Dati su i najelementarniji delovi sintakse jezika C++ koji omogućavaju formiranje klasa. Kao što je rečeno, osim podataka u sadržaj klasa (samim tim i objekata) spadaju i funkcije - funkcije članice. Opšta pravila za uključivanje funkcija u klase su:

1. Unutar klase navodi se prototip funkcije (deklaracija tipa funkcije, ime funkcije, tipovi parametara funkcije)

```
class ime_klase
{
          tip podatak1;
          tip podatak2;
          ...
          tip ime_funkcije1(tipovi_parametara);
          tip ime_funkcije2(tipovi_parametara);
          ...
};
```

2. U programu se definiše kompletna funkcija koja pripada konkretnoj klasi:

```
tip ime_klase :: ime_funkcije( parametri )
{
          // sadrzaj - programski kod funkcije
}
```

3. Deklarisanje objekta izvodi se kao i za svaki drugi tip podataka:

```
ime_klase ime_objekta;
```

4. Poziv funkcije članice objekta se izvodi na isti način na koji se pristupa podacima objekta:

```
ime_objekta.ime_promenljive;

ime_objekta.ime_funkcije(parametri);
```

Funkcije članice klase su one funkcije čija se definicija ili prototip nalazi u okviru definicije (tela) klase. One mogu da operišu sa svim objektima klase čiji su član, i imaju pristup svim ostalim podacima i funkcijama članicama klase tog objekta.

PRIMER DEFINICIJE I UPOTREBE FUNKCIJA ČLANICA KLASE

Najčešća praksa je da se deklaracija funkcije članice navede u okviru definicije klase, a da se implementacija izvrši van tela klase i to u nekom drugom fajlu

Uzmimo prethodno pomenutu klasu Box, i u njoj napišimo definiciju funkcije koja će imati pristup ostalim članicama klase, umesto da tim članovima pristupamo direktno:

```
class Box
{
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box
      double getVolume(void);// Returns box volume
};
```

Funkcije članice mogu biti definisane ili u okviru tela klase, ili van tela klase korišćenjem operatora pristupa (scope resolution operator), :: . Definicijom funkcije unutar klase ona automatski postaje linijska, odnosno inline funkcija, čak iako se ne navede ključna reč inline. Stoga je moguće funkciju za određivanje zapremine Volume() definisati korišćenjem operatora pristupa scope resolution operator, :: , van tela klase:

```
double Box::getVolume(void)
{
    return length * breadth * height;
}
```

Ili unutar klase na sledeći način:

```
class Box
{
   public:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box

      double getVolume(void)
      {
         return length * breadth * height;
      }
};
```

U prethodnom primeru, jedina bitna stvar o kojoj treba obratiti pažnju je da treba navesti ime klase ispred operatora ::. Funkcija članica klase može biti pozvana korišćenjem operatora tačke (.) uz odgovarajući objekat te klase, kao što je opisano u sledećem primeru:

```
Box myBox;          // Create an object

myBox.getVolume();  // Call member function for the object
```

UVOD U FUNKCIJE PRISTUPA ČLANOVIMA KLASE - GETER I SETER METODE

U cilju sakrivanja podaci članovi klase se uglavnom definišu kao privatni, nedostupni korisnicima klase. Zatim se u svrhu pristupa i izmena podataka koriste geter i seter metode

Iskoristimo prethodni koncept u cilju postavljanja i dobijanja (set i get) vrednosti različitih podataka članova klase Box:

```
#include <iostream>
using namespace std;
class Box
{
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box

      // Member functions declaration
      double getVolume(void);
      void setLength( double len );
      void setBreadth( double bre );
      void setHeight( double hei );
};

// Member functions definitions
double Box::getVolume(void)
{
    return length * breadth * height;
}

void Box::setLength( double len )
{
    length = len;
}

void Box::setBreadth( double bre )
 breadth = bre;
}
```

```
void Box::setHeight( double hei )
{
    height = hei;
}

// Main function for the program
int main( )
{
   Box Box1;                // Declare Box1 of type
Box
   Box Box2;                // Declare Box2 of type
Box
   double volume = 0.0;     // Store the volume of a
box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   return 0;
}
```

UVOD U FUNKCIJE PRISTUPA ČLANOVIMA KLASE - GETER I SETER METODE

U cilju sakrivanja podaci članovi klase se uglavnom definišu kao privatni, nedostupni korisnicima klase. Zatim se u svrhu pristupa i izmena podataka koriste geter i seter metode

Rezultat prethodnog programa biće:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Konvencija je da imena klasa počinju velikim slovom. Evo primera gde se definiše metoda lajati() za klasu Pas, i gde se ova metoda poziva u glavnom programu:

```
#include <string>
#include<iostream>
using namespace std;
class Pas
{
public:
           int starost;
           int tezina;
           string boja;
           void lajati();
};
void Pas::lajati()
{
           cout << "Av, av" << endl;
}
int main()
{
           Pas Jimi;
           Jimi.lajati();
           return 0;
}
```

LINIJSKE (INLINE) FUNKCIJE

Da bi kreirali linijsku funkciju koristimo ključnu reč inline ispred naziva funkcije. Najčešće se za linijske funkcije proglašavaju one koji imaju mali broj instrukcija u telu funkcije

Linijske funkcije su jedan moćan koncept koji se često koristi kod klasa. Ukoliko je neka funkcija definisana kao inline, onda će kompajler u toku kompajliranja zameniti svaki poziv funkcije sa kodom koji se nalazi unutar tela inline (linijske) funkcije.

Svaka promena linijskih funkcija će stoga zahtevati ponovno rekompajliranje delova koda koji pozivaju linijsku funkciju jer kompajler mora da zameni sve delove koda ponovo, u suprotnom će se zadržati stara funkcionalnost.

Kao što nam je već poznato, da bi kreirali linijsku funkciju koristimo ključnu reč inline ispred naziva funkcije a definiciju funkcije treba napisati pre prvog poziva te funkcije. Kompajler može ignorisati ključnu reč inline ukoliko funkcija ima više od jedne linije. Ono što treba napomenuti je da sve funkcije članice čija se definicija nalazi unutar tela klasa automatski postaju linijske iako se ne navede specifikator inline. Primer:

```
class Friend
{
public:
          void setAge (int a) {age = a;};
          int getAge() {return age;};
private:
          string surname; int phone; int age;
};
```

Funkcije sa svega nekoliko linija koda mogu se tretirati kao linijske primenom instrukcije inline, u cilju da se izbegne skakanje naokolo što usporava program. Primer:

```
inline void People::setPhone(int y) {phone = y;}
```

Metode klase koje imaju mali broj instrukcija mogu se deklarisati kao inline, na isti način kako smo ranije obične funkcije deklarisali kao inline. U nastavku je dat primer:

```
class Pas
{
          ...
          void setStarost(int godine);
          ...
          void lajati()
{cout<<"Av,av"<<endl;};//automatski inline
          ...
};

...
inline void Pas::setStarost(int godine)
{
          starost = godine;
}
...
int main()
{
          ...
}
```

Konstruktor i destruktor

konstruktor, tipovi konstruktora destruktor, copy konstruktor

- *Konstruktori*
- *Destruktor*
- *Konstruktor kopiranja*

03

UVODNA RAZMATRANJA

Konstruktor je funkcija članica koja ima isto ime kao i klasa. Poziva se pri kreiranju objekta, dok se pri uništenju objekta poziva takođe funkcija istoga naziva kao i klasa koja se zove destruktor

U okviru ove sekcije upoznaćemo se sa osnovnim stvarima u vezi konstruktora, destruktora i konstrukora kopiranja.

Konstruktor je funkcija članica koja ima isto ime kao i klasa. Uloga konstruktora je da izvrši inicijalizaciju podataka članova klase. U nastavku će biti opisani podrazumevajući (default) i konstruktor sa parametrima (overloaded konstruktor).

Za razliku od programskog jezika Java, gde se takozvani „objekti otpaci“ automatski brišu korišćenjem takozvanog garbage collector-a, u C++-u je neophodno napisati funkciju koja mora da obriše alocirani memorijski prostor iz hip memorije. Podrazumevajuće funkcije članice klase koje služe u ovu svrhu se nazivaju destruktori i o njima će biti nešto više reči u nastavku.

C++ ima mogućnost da se pri pozivu konstruktora izvrši takozvano kopiranje objekata. Tipovi konstruktora koji imaju mogućnost izvršenja ovakvih operacija se nazivaju konstruktori kopiranja.

Konstruktori

konstruktor, podrazumevajući i konstruktor s parametrima

- *Definicija konstruktora*
- *Primer upotrebe konstruktora*
- *Konstruktor sa parametrima*
- *Korišćenje liste za inicijalizacuju*

03

DEFINICIJA KONSTRUKTORA

Konstruktor klase je specijalna funkcija članica klase koja se izvršava svaki put kada se kreira nova instanca neke klase

U prethodnom primeru kod klase Box sastavni deo programa su bile i funkcije za setovanje vrednosti tj. za inicijalizaciju podataka članova objekta. Kao i svaka funkcija članica i ova funkcija mora biti pozvana iz programa da bi se izvršila. Analizom većeg broja objekata može se zaključiti da većina od njih zahteva inicijalizaciju podataka koji im pripadaju. Ne primer, ako je objekat realni niz klase matrica onda ga pri deklarisanju obično treba i unuliti. Osim toga, obzirom na proizvoljnost dimenzije niza, potrebno je izvršiti dinamičko alociranje memorije za niz. Ovakve i slične potrebe u sintaksi C++ jezika podržavaju specijalne funkcije članice klasa koje se zovu konstruktori.

Konstruktor klase je specijalna funkcija članica klase koja se izvršava svaki put kada se kreira nova instanca neke klase. Kontruktor će imati isti naziv kao i sama klasa, dok povratni tip funkcije neće biti čak ni void. Konstruktori su veoma korisni u cilju postavljanja početnih (inicijalnih) vrednosti određenim podacima članovima.

Sintaksa za formiranje konstruktora se svodi na pisanje funkcije čije je ime isto kao i klasa na koju se konstruktor odnosi. Parametri se navode sa tipovima, na uobičajen način. Treba naglasiti da konstruktor funkcija ne vraća nikakvu vrednost, čak ni void, pa se zato pri pisanju funkcije piše samo njeno ime bez tipa. Opšti oblik za konstruktor je:

```
ime_klase :: ime_klase ( parametri )
{
          // sadrzaj - programski kod funkcije
}
```

PRIMER UPOTREBE KONSTRUKTORA

Konstruktor klase ne vraća nikakvu vrednost, čak ni void, pa se zato pri pisanju konstruktora funkcije piše samo njegovo ime bez tipa

Sledeći primer demonstrita upotrebu konstruktora:

```
#include <iostream>
using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();  // This is the constructor

   private:
      double length;
};

// Member functions definitions including
constructor
Line::Line(void)
{
    cout << "Object is being created" << endl;
}
```

Funkcije getLength() i setLength() su već definisane u prethodnom primeru i možemo iskoristiti to i u ovom primeru.

Glavni program može biti napisan na sledeći način:

```
// Main function for the program
int main( )
{
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength()
<<endl;

   return 0;
}
```

Nakon kompajliranja i izvršavanja prethodnog koda dobija se sledeći rezultat:

```
Object is being created
Length of line : 6
```

KONSTRUKTOR SA PARAMETRIMA

Konstruktor sa parametrima omogućava da se kroz listu argumenata konstruktora dodele inicijalne vrednosti članovima objekta u samom trenutku kreiranja objekta

Podrazumevajući konstruktor je funkcija koja nema argumente ali, ako za tim postoji potreba, može se definisati konstruktor koji ima parametre. Konstruktor sa parametrima pomaže da se dodele inicijalne vrednosti članovima objekta u samom trenutku kreiranja objekta, kao što je pokazano sledećim primerom:

```
#include <iostream>
using namespace std;
class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line(double len);  // This is the constructor

   private:
      double length;
};

// Konstruktor sa parametrima i ostale funkcije
clanice
Line::Line( double len)
{
    cout << "Object is being created, length = " <<
len << endl;
    length = len;
}

void Line::setLength( double len )
{
    length = len;
}
```

Glavni program možemo napisati na sledeći način:

```
// Main function for the program
int main( )
{
   Line line(10.0);

   // get initially set length.
   cout << "Length of line : " << line.getLength()
<<endl;
   // set line length again
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength()
<<endl;

   return 0;
}
```

Rezultat programa će biti:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

KORIŠĆENJE LISTE ZA INICIJALIZACUJU

Lista za incizijalizaciju se ubacuje nakon parametara konstruktora, i znaka dvotačke (:), gde se navodi svaka pojedinačna vrednost za promenljive članice, koje su razdvojene znakom zarez (,)

C++ sadrži drugačiji način inicijalizacije članova objekata koji omogućava da se vrednosti članova objekta inicijalizuju u trenutku kreiranja. Ovo se ostvaruje korišćenjem liste za inicijalizaciju. U prvoj lekciji o C++-u, videli smo da je moguće dodeliti vrednost nekoj promenljivoj na dva načina: eksplicitni i implicitni:

```
int nValue = 5; // explicit assignment
double dValue(4.7); // implicit assignment
```

Korišćenje liste za inicijalizaciju je veoma slično implicitnoj dodeli vrednosti. U slučaju konstruktora sa parametrima, moguće je koristiti sledeću sintaksu za inicijalizaciju polja, odnosno članova, objekata:

```
Line::Line( double len): length(len)
{
    cout << "Objekat kreiran, len = " << len <<
endl;
}
```

Prethodna sintaksa je identična sintaksi koja sledi:

```
Line::Line( double len)
{
    cout << "Objekat kreiran, len = " << len <<
endl;
    length = len;
}
```

Lista za inicijalizaciju se ubacuje odmah nakon navođenja parametara konstruktora, i znaka dvotačke (:), gde se navodi svaka pojedinačna vrednost za promenljive članice, koje su razdvojene znakom zarez. Može se primeti da u ovom slučaju više ne moramo da vršimo eksplicitnu dodelu vrednosti u telu konstruktora, pošto je taj deo koda zamenjem listom za inicijalizaciju. Takođe treba naglasiti da se lista za inicijalizaciju ne završava znakom tačka-zarez. Ukoliko npr. kod neke klase C imate potrebu da inicijalizujete višestruki broj polja kao npr. X, Y, Z, itd., onda je moguće koristiti istu sintaksu kao u prethodnom primeru, s tim što će polja biti odvojena zarezima, kao što je prikazano u nastavku:

```
C::C(double a, double b, double c): X(a), Y(b), Z(c)
{
  ....
}
```

Preporuka je da se koristi ova nova sintaksa (čak i kada se ne koriste promenljive članice koje su ili reference ili konstante) kao kod liste za inicijalizaciju, što je poželjno kod kompozicije i nasleđivanja (što će biti opisano u nekoj od narednih lekcija).

Destruktor

destruktor, definicija destruktora, primena destruktora

- *Definicija destruktora klase*
- *Primer upotrebe destruktora*
- *Primer upotrebe konstruktora i destruktora*

03

DEFINICIJA DESTRUKTORA KLASE

Destruktori su funkcije koje se, kao i konstruktori, pozivaju automatski, ali u ovom slučaju pri prestanku postojanja objekta tj. pri njegovom destruktuiranju

Komplementarnu ulogu konstruktorima u C++ programima imaju destruktori. Destruktori su funkcije koje se, kao i konstruktori, pozivaju automatski, ali u ovom slučaju pri prestanku postojanja objekta tj. pri njegovom destruktuiranju. Najčešći zadatak destruktora je dealociranje memorije i reinicijalizacija promenljivih (npr. brojača objekata određene klase). U programu se destruktor funkcije lako uočavaju jer im ime počinje znakom "~" (tilda), a sledi naziv klase kojoj destruktor pripada. Destruktor, kao i konstruktor, ne vraća nikakvu vrednost, pa se deklariše bez navođenja tipa. Opšti oblik destruktora je:

```
ime_klase :: ~ime_klase ( parametri )
{
          // sadrzaj - programski kod funkcije
}
```

Destruktor je specijalna funkcija članica klase koja se izvršava svaki put kada objekat klase za koju je definisan, izlazi iz opsega važenja ili prilikom korišćenja operatora delete nad pokazivačem, koji pokazuje na objekat te klase (o pokazivačima na objekte će biti više reči u nekoj od narednih lekcija).

Destruktor je od velike koristi za oslobađanje resursa pre izlaska iz programa, kao što je na primer zatvaranja fajlova, oslobađanje dinamički zauzete memorije itd.

PRIMER UPOTREBE DESTRUKTORA

U programu se destruktor funkcije lako uočavaju jer im ime počinje znakom "~" (tilda), a sledi naziv klase kojoj destruktor pripada

Sledeći primer demonstrira koncept i primenu destruktora:

```
#include <iostream>
using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // This is the constructor
declaration
      ~Line();  // This is the destructor:
declaration

   private:
      double length;
};

// Member functions definitions including
constructor
Line::Line(void)
{
    cout << "Object is being created" << endl;
}
Line::~Line(void)
{
    cout << "Object is being deleted" << endl;
}
```

Glavni program možemo napisati na sledeći način:

```
// Main function for the program
int main( )
{
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength()
<<endl;

   return 0;
}
```

Nakon izvršenja programa, dobićemo sledeći izlaz:

```
Object is being created
Length of line : 6
Object is being deleted
```

PRIMER UPOTREBE KONSTRUKTORA I DESTRUKTORA

Destruktor je od velike koristi za oslobađanje resursa pre izlaska iz programa, kao što je na primer zatvaranja fajlova, oslobađanje dinamički zauzete memorije itd.

U nastavku je dat primer definisanja konstruktora i destruktora za klasu Pas, i upotrebe konstruktora da se istovremeno i kreira i inicijalizuje objekat po imenu Jimi, i to elegantno, jednom instrukcijom, bez višestruke upotrebe tačka-operatora

```
#include <string>
#include<iostream>
using namespace std;

class Pas
{
public:
            Pas(int inicStarost, int inicTezina,
string inicBoja);//konstruktor
            ~Pas();//destruktor
            int starost;
            int tezina;
            string boja;
            void lajati();
};

Pas:: Pas(int inicStarost, int inicTezina, string
inicBoja)
{
            starost = inicStarost;
            tezina = inicTezina;
            boja = inicBoja;
}

Pas::~Pas()
{
}
```

Funkciju članicu lajati() i glavnu main() funkciju možemo napisati na sledeći način

```
void Pas::lajati()
{
            cout << "Av, av" << endl;
}

int main()
{
            Pas Jimi(3, 15, "crna");
            cout << Jimi.starost << "," <<
Jimi.tezina << ","
                        << Jimi.boja << endl;
            return 0;
}
```

Kao rezultat rada programa, na ekranu se pojavljuje:

3,15, crna

Konstruktor kopiranja

Konstruktor kopiranja, običan konstruktor

- *Definicija konstruktora kopiranja*
- *Primeri upotrebe običnog i konstruktora kopiranja*

03

DEFINICIJA KONSTRUKTORA KOPIRANJA

Konstruktor kopiranja je konstruktor koji kreira objekat i inicijalizuje ga vrednostima objekta iste klase, koji je prethodno kreiran

Konstruktor kopiranja je konstruktor koji kreira objekat i inicijalizuje ga vrednostima objekta iste klase, koji je prethodno kreiran. Konstruktor kopiranja se koristi u nekoliko slučajeva, i to:

- da inicijalizuje jedan objekat korišćenjem već postojećeg objekta istog tipa.
- da kopira jedan objekat kako bi kopiju prosledio funkciji.
- da kopira jedan objekat koji će biti vraćen kao rezultat iz funkcije

Ukoliko konstruktor kopiranja nije definisan u okviru klase, kompajler ga automatski sam definiše. Ukoliko klasa ima pokazivačku promenljivu članicu, i ukoliko se ta promenljiva koristi za dinamičko alociranje memorije onda je neophodno da postoji konstruktor kopiranja. Najuobičajeni oblik konstruktora kopiranja je prikazan u sledećem primeru:

```
classname (const classname &obj) {
   // body of constructor
}
```

gde je **obj** referenca objekta koji se koristi za inicijalizaciju drugog objekta. U nastavku je dat primer definicije klase Line i definisanja osnovnog i konstruktora kopiranja:

```
#include <iostream>
using namespace std;

class Line
{
   public:
      int getLength( void );
      Line( int len );             // simple
constructor
      Line( const Line &obj);  // copy constructor
      ~Line();                      // destructor

   private:
      int *ptr;
};

// Member functions definitions including
constructor
Line::Line(int len)
{
    cout << "Normal constructor allocating ptr" <<
endl;
    // allocate memory for the pointer;
    ptr = new int;
    *ptr = len;
}

Line::Line(const Line &obj)
{
    cout << "Copy constructor allocating ptr." <<
endl;
    ptr = new int;
   *ptr = *obj.ptr; // copy the value
}
```

DEFINICIJA KONSTRUKTORA KOPIRANJA

Konstruktor kopiranja je konstruktor koji kreira objekat i inicijalizuje ga vrednostima objekta iste klase, koji je prethodno kreiran

Konstruktor kopiranja je konstruktor koji kreira objekat i inicijalizuje ga vrednostima objekta iste klase, koji je prethodno kreiran. Konstruktor kopiranja se koristi u nekoliko slučajeva, i to:

- da inicijalizuje jedan objekat korišćenjem već postojećeg objekta istog tipa.
- da kopira jedan objekat kako bi kopiju prosledio funkciji.
- da kopira jedan objekat koji će biti vraćen kao rezultat iz funkcije

Ukoliko konstruktor kopiranja nije definisan u okviru klase, kompajler ga automatski sam definiše. Ukoliko klasa ima pokazivačku promenljivu članicu, i ukoliko se ta promenljiva koristi za dinamičko alociranje memorije onda je neophodno da postoji konstruktor kopiranja. Najuobičajeni oblik konstruktora kopiranja je prikazan u sledećem primeru:

```
classname (const classname &obj) {
   // body of constructor
}
```

gde je **obj** referenca objekta koji se koristi za inicijalizaciju drugog objekta. U nastavku je dat primer definicije klase Line i definisanja osnovnog i konstruktora kopiranja:

```
Line::~Line(void)
{
    cout << "Freeing memory!" << endl;
    delete ptr;
}
int Line::getLength( void )
{
    return *ptr;
}

void display(Line obj)
{
   cout << "Length of line : " <<
obj.getLength() <<endl;
```

PRIMERI UPOTREBE OBIČNOG I KONSTRUKTORA KOPIRANJA

Ukoliko klasa ima pokazivačku promenljivu članicu, i ukoliko se ta promenljiva koristi za dinamičko alociranje memorije onda je neophodno da postoji konstruktor kopiranja

U cilju boljeg uvida u to kako radi kontruktor kopiranja analizaćemo dva sledeća primera. Pretpostavimo da je u prvom primeru glavni program napisan na sledeći način:

```
// Main function for the program
int main( )
{
   Line line(10);

   display(line);

   return 0;
}
```

Nakon kompajliranja i izvršavanja prethodnog programa dobiće se sledeći izlaz:

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
```

Pogledajmo sada primer sa malim izmenama u cilju kreiranja novog objekta korišćenjem već napravljenog objekta istog tipa:

```
#include <iostream>
using namespace std;

/* definicija klase i funkcija klase je ostala
ista*/

// Main function for the program
int main( )
{
   Line line1(10);
   Line line2 = line1; // Also calls copy
constructor

   display(line1);
   display(line2);

   return 0;
}
```

Rezultat programa će sada biti:

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
```

Pokazivač this

Pokazivač this, skriveni this pokazivač, objekti i adrese

- *Osnovi o pokazivaču this*
- *Upotreba pokazivača this*

04

OSNOVI O POKAZIVAČU THIS

Svaki objekat u C++-u ima pristup svojoj adresi (adresi na kojoj je smešten memoriji) pomoću jednog veoma bitnog pokazivača koji se naziva pokazivač this

Svaki objekat ima jedan specijalni *pokazivač* pod nazivom this koji pokazuje na sam objekat. Članovima objekta klase Friend se može pristupiti pomoću ovog pokazivača, kao npr:

```
this->age;
this->phone;
this->surname;
```

Svaki objekat u C++-u ima pristup svojoj adresi (adresi na kojoj je smešten u memoriji) pomoću jednog veoma bitnog pokazivača koji se naziva pokazivač this. Pokazivač this je jedan implicitni parametar za sve funkcije članice klase. Stoga, unutar tela funkcije članice, pokazivač this se može koritiiti za objekat nad kojim je funkcija članica i pozvana.
Prijateljske funkcije nemaju ovaj pokazivač this, jer prijateljske funkcije nisu članice klase. Samo funkcije članice imaju pokazivač this. O prijateljskim funkcijama će biti reči u sledećoj lekciji.
Korišćenje pokazivača this može biti korisno kod definisanja pristupnih funkcija, gde argument često može da ima isto ime kao i član klase. Onda pokazivač this može da omogući razliku između argumenta i člana klase. Pokazivač this se primenljuje u sledećem primeru:

```
void Friend::setAge(int age)
{
            this -> age = age;
}
void Friend::setPhone(int phone)
{
            this -> phone = phone;
}
void Friend::setSurname(string surname)
{
            this -> surname = surname;
}
```

Nekonfliktna funkcija zadavanja:

Ako ime člana i ime argumenta u pristupnoj funkciji nisu u suprotnosti, pokazivač this nije potreban. Primer:

```
void Friend::setAge(int ageValue)
{
            age = ageValue;
}
```

UPOTREBA POKAZIVAČA THIS

Unutar tela funkcije članice, pokazivač this se može koritiiti za objekat nad kojim je funkcija članica i pozvana

Pogledajmo i isprobajmo sledeći primer da bi smo bolje razumeli korišćenje pokazivača this:

```
#include <iostream>
using namespace std;

class Box
{
   public:
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      int compare(Box box)
      {
         return this->Volume() > box.Volume();
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
```

Glavni program možemo napisati na sledeći način:

```
int main(void)
{
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   if(Box1.compare(Box2))
   {
      cout << "Box2 is smaller than Box1" <<endl;
   }
   else
   {
      cout << "Box2 is equal to or larger than Box1"
<<endl;
   }
   return 0;
}
```

Rezultat programa će biti:

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

Odvojena definicija klase

Odvojena definicija, interfejs klase, header fajlovi

- *Osnovna razmatranja*
- *Primer odvojene definicije klase*

05

OSNOVNA RAZMATRANJA

Odvajanjem definicije klase u fajl zaglavlja veličina glavnog programa se smanjuje i klasa postaje modularna pa se može koristiti unutar bilo kojeg programa pomoću #include direktive

U prethodnim primerima smo definicije odnosno deklaracije klase uglavnom stavljali u isti fajl sa glavnim programom, i to smo radili zbog postizanja jednostavnosti pisanja programa.

Međutim, u praksi je drugačije. Većina programera piše deklaracije klasa odvojeno u *header* fajlu. Ovaj fajl, obično, ima isto ime kao i fajl sa glavnim programom ali ima ekstenziju .h (ili .hpp) dok glavni fajl ima ekstenziju .cpp. Fajl zaglavlja, *header file*, koji sadrži deklaraciju klase se može zatim uključiti u glavnom programu korišćenjem jedne #include pretprocesorske direktive. Ovo je slično sa uključivanjem neke klase iz standardne biblioteke Standard C++ Library, npr. klase <string> (instrukcija #include<string>), ali ime *header* fajla, zajedno sa svojom ekstenzijom, mora biti ubačeno između duplih znakova navoda (#include "Klasa.h"), umesto ugaonih zagrada.

Odvajanjem deklaracije klase na ovaj način, veličina glavnog programa se smanjuje, i klasa postaje modularna pa se može koristiti unutar bilo kojeg programa, jednostavno pomoću dodavanja #include direktive. Međutim, u slučaju da drugi programer hoće da dotičnu odvojenu klasu uključi u svoj program, treba da zna imena metoda klase i listu argumenata.

PRIMER ODVOJENE DEFINICIJE KLASE

U slučaju da drugi programer želi da dotičnu odvojenu klasu uključi u svoj program, treba da zna imena metoda klase i listu argumenata

Evo primera, gde se koristi klasa Pas, i u glavnom programu je postavljena #include direktiva za uključivanje *header* fajla, koji se odvojeno formira sa ekstenzijom .h. Fajl sa deklaracijom klase, pas.h fajl je

```
#include<string>
#include<iostream>
using namespace std;
class Pas
{
public:
            Pas(int inicStarost, int inicTezina,
string inicBoja)
            {
                        starost = inicStarost;
                        tezina = inicTezina;
                        boja = inicBoja;
            }
            ~Pas(){}
            void lajati()
            {
                        cout<<"Av, av"<<endl;
            }
            int starost;
            int tezina;
            string boja;
};//tacka-zarez na kraju
```

Zatim, imamo pas.cpp fajl, sa #include "pas.h" direktivom,

```
#include "pas.h"
int main()
{
            Pas Jimi(3, 15, "crna");
            Pas Hilari(4, 10, "braon");
            Jimi.lajati();
            Hilari.lajati();
            cout<<Jimi.starost<<","<<Jimi.tezina<<",
"<<Jimi.boja<<endl;
            cout<<Hilari.starost<<","<<Hilari.tezina
<<","<<Hilari.boja<<endl;
            return 0;
}
```

Kao rezultat rada programa, na ekranu se pojavljuje:

3,15,crna

4,10,braon

Primetimo da su instrukcije:

```
#include<string>
#include<iostream>
using namespace std;
```

uključene u .h fajl pa ih nije potrebno stavljati u .cpp fajl. Takođe, primetimo da su metode klase stavljene unutar klase, pa ne treba koristiti operator „::“ kod njihovog definisanja.

Područje (oblast važenja) klase

Oblast klase, class scope, podaci članice, funkcije članice

- *Oblast važenja klase*
- *Oblast važenja kod definicije klase u odvojenom fajlu*

06

OBLAST VAŽENJA KLASE

Pri pisanju složenog projekta preporuka je da deklaracija klase bude u header fajlu, definicija funkcija u .cpp fajlu istog naziva, a da se u fajl sa glavnim programom uključi header fajl klase

Prilikom pisanja složenog programa koji sadrži klase može se koristiti sledeća metodologija:

- Razmisliti koje klase bi bile poželjne, i koje podatke i funkcije ove klase treba da obuhvataju.
- Deklarisati klasu u datoteci sa zaglavljima (header file), npr. klasa deklarisana u myclass.h.
- Definisati funkcije članove kao izvornu datoteku zajedno sa glavnim programom, npr. myclass.cpp.
- Koristiti #include po potrebi u programima , npr: #include "myclass.h".

Oblast klase (class scope):

Podaci članovi (*data members*) klase su automatski u području važenja (*scope*) funkcija članova (*member functions*) klase. Stoga podatke članove nije potrebno navoditi kao parametre funkcija članova ili kao povratne vrednosti funkcija članova.

Primer:

Fajl triangle.h:

```
// DEKLARACIJA KLASE:By convention, a class name starts with a capital
class Trougao {
public:
          double s1, s2, a12; //Here are three data members of the class
          Trougao (double, double, double); //Here is the constructor
          //Here are declarations ('prototypes') of member functions ('methods') of the class
          double visina(); double povrs (); double strana3();
private: //The private methods can only be accessed within the class
          double ugao13(); double ugao23();
};
```

OBLAST VAŽENJA KOD DEFINICIJE KLASE U ODVOJENOM FAJLU

Podaci članovi klase su automatski u području važenja funkcija članova klase. Stoga podatke članove nije potrebno navoditi kao parametre ili kao povratne vrednosti funkcija članova

Da bi smo stekli bolji uvid u područje klase, nastavljamo sa definisanjem klase prethodnog primera. Definicija klase će biti smeštena u zasebnom fajlu triangle.cpp:

```
#include "triangle.h"
#include "math.h"

#define PI 3.14
//KONSTRUKTOR:
//This function concerns Trougao namespace (i.e.,
Trougao class)
//A constructor function has the same name as the
class it instantiates, and no return value
Trougao::Trougao (double strana1, double strana2,
double ugao12)
{ // duza strana je s1
 if (strana1 > strana2)
 {
  s1 = strana1;
  s2 = strana2;
 }
 else
 {
  s1 = strana2;
  s2 = strana1;
 }
 if (ugao12 < 0)
  ugao12 = -1 * ugao12;
 // a12 in degrees, converted to radians
 a12 = ugao12 * PI / 180;
}
```

```
//OSTALE FUNKCIJE CLANOVI:
double Trougao::visina()
{
 return s2 * sin(a12);
}// in the scope of Trougao

double Trougao::povrs()
{
 return 0.5 * s1 * visina();
}

double Trougao::strana3()
{
 // TODO:
 return 0.5 * s1 * visina();
}
//Because the function is in the scope of Triangle,
the compiler 'knows' that the s1 referred to
```

OBLAST VAŽENJA KOD DEFINICIJE KLASE U ODVOJENOM FAJLU

Podaci članovi klase su automatski u području važenja funkcija članova klase. Stoga podatke članove nije potrebno navoditi kao parametre ili kao povratne vrednosti funkcija članova

Kao što možemo primetiti iz prethodnog koda, iako je konstruktor klase definisan u zasebnom fajlu triangle.cpp, pristup podacima te iste klase se vrši samo navođenjem podataka članova (npr. s1 i s2), čime se direktno vrši izmena stvarnih podataka kreiranog objekta. U cilju testiranja kreirane klase ostaje nam da napišemo glavni pokretački program, i njega ćemo smestiti u fajlu main.cpp:

```
#include <iostream>
#include "triangle.h"

using namespace std;
int main()
{
              double side1, side2, a12;
              cout << "ukucaj duzine 2 strane trougla\n";
              cin >> side1;
              cin >> side2;
              cout << "ukucaj ugao izmedju njih
(degrees)\n";
              cin >> a12;
              // Create an instance of Trougao : my_tro
              // Initialise with strana1, strana2 and a12
              Trougao my_tro(side1, side2, a12);
              cout << "duyina trece strane " <<
my_tro.strana3() << endl;
              cout << "povrsina " << my_tro.povrs() <<
endl;
}
```

Konstantne funkcije i objekti

konstantne funkcije, konstantni objekti

- *Konstantni (const) objekti*
- *Konstantne funkcije članice klase*
- *Pravilno korišćenje konstantnih funkcija*
- *Primeri upotrebe konstantnih funkcija*

07

KONSTANTNI (CONST) OBJEKTI

Jednom kad je konstantan objekat klase inicijalizovan pomoću konstruktora, bilo kakav pokušaj promene sadržaja obekta biće nedozvoljena operacija.

U prethodnoj lekciji gde smo imali prosleđivanje parametara po referenci opisali smo značaj korišćenja konstatnih argumenata. Da se podsetimo, definisanjem konstantnog parametra funkcije sprečavamo nepredviđene promene koje mogu da se reflektuju na vrednost stvarnog argumenta.

Baš kao i pri radu sa primitivnim tipovima podataka (int, double, char, itd…), tako i objekte klase možemo proglasiti za konstantne korišćenjem ključne reči const. Sve konstantne promenljive moraju biti inicijalizovane u trenutku kreiranja. U slučaju primitivnih tipova podataka, inicijalizacija se može izvršiti implicitnom ili eksplicitnom dodelom:

```
const int nValue = 5; // initialize explicitly
const int nValue2(7); // initialize implictly
```

U slučaju klasa, inicijalizacija se ostvaruje korišćenjem konstruktora:

```
const Date cDate; // initialize using default constructor
const Date cDate2(10, 16, 2020); // initialize using
parameterized constructor
```

Ukoliko klasa nije inicijalizovana korišćenjem konstruktora sa parametrima, javni podrazumevajući konstruktor mora biti definisan. U slučaju da ne postoji podrazumevajući konstruktor pojaviće se kompajlerska greška.

Jednom kad je konstantan objekat klase inicijalizovan pomoću konstruktora, bilo kakav pokušaj promene sadržaja objekta biće nedozvoljena operacija. Ovo uključuje ili direktno menjanje članova ili posredno pozivom neke od funkcije članice:

```
class Something
{
public:
    int m_nValue;

    Something() { m_nValue = 0; }
    void ResetValue() { m_nValue = 0; }
    void SetValue(int nValue) { m_nValue = nValue; }
    int GetValue() { return m_nValue; }
};
int main()
{
    const Something cSomething; // calls default
constructor
     cSomething.m_nValue = 5; // violates const
    cSomething.ResetValue(); // violates const
    cSomething.SetValue(5); // violates const
     return 0;
}
```

Sve tri prethodne linije koda koje uključuju rad nad instancom cSomething su ilegalne jer narušavaju konstantnost promenljive cSomething pokušajima da promene vrednost podatka člana klase ili pozivima funkcija članica koje takođe pokušavaju da promene vrednost nekog od podatka pobjekta.

KONSTANTNE FUNKCIJE ČLANICE KLASE

Konstantna funkcija članica je ona koja garantuje da neće promeniti vrednosti bilo kog podatka klase, niti da će se iz nje pozvati funkcije koje nisu deklarisane kao konstantne

Sada, uzmimo u obzir sledeći poziv funkcije u prethodnom programu:

```
std::cout << cSomething.GetValue();
```

Iznenađujuće je to da će se javiti kompajlerska greška! Razlog je taj što se nad konstantnim objektima mogu pozivati samo one funkcije koje su deklarisane kao konstantne, a funkcija GetValue() nije označena kao konstantna. **Konstantna funkcija članica** je ona koja garantuje da neće promeniti vrednosti bilo kog podatka klasa, niti da će se iz nje pozvati funkcije koje nisu deklarisane kao konstantne. Da bi smo neku funkciju načinili konstantnom funkcijom, kao u ovom slučaju funkciju GetValue(), neophodno je samo dodati ključnu reč const u prototip funkcije, kao što je pokazano sledećim primerom:

```
class Something
{
public:
    int m_nValue;

    Something() { m_nValue = 0; }

    void ResetValue() { m_nValue = 0; }
    void SetValue(int nValue) { m_nValue = nValue; }

    int GetValue() const { return m_nValue; }
};
```

Sada smo funkciju GetValue() načinili konstantnom funkcijom klase, tako da nju možemo pozvati nad bilo kojim konstantnim objektom. Konstantne funkcije klase, deklarisane van bloka (definicije) klase moraju sadržati ključnu reč const i kod deklaracije u okviru tela klase i kod definicije funkcije koja se nalazi van bloka klase:

```
class Something
{
public:
    int m_nValue;

    Something() { m_nValue = 0; }

    void ResetValue() { m_nValue = 0; }
    void SetValue(int nValue) { m_nValue = nValue; }

    int GetValue() const;
};

int Something::GetValue() const
{
    return m_nValue;
}
```

PRAVILNO KORIŠĆENJE KONSTANTNIH FUNKCIJA

Svaka konstantna funkcija članica koja pokuša da promeni vrednost podatka klase ili da pozove neku drugu funkciju članicu koja nije konstantna će da proizvede kompajlersku grešku

Svaka konstantna funkcija članica koja pokuša da promeni vrednost podatka klase ili da pozove neku drugu funkciju članicu koja nije konstantna će da proizvede kompajlersku grešku, kao u sledećem primeru:

```
class Something
{
public:
    int m_nValue;

    void ResetValue() const { m_nValue = 0; }
};
```

U ovom primeru funkcija ResetValue() je definisana kao konstantna funkcija, ali ona pokušava da promeni sadržaj podatka m_nValue klase Something, što će kao posledicu izazvati kompajlersku grešku.
Treba napomenuti, što se i podrazumeva, da konstruktori nikako i ne treba da budu definisani kao konstantni, zbog već dobro poznatog razloga.
Konačno, iako se ovo ne praktikuje često, moguće je da se izvrši preklapanje funkcija (overload) tako da imamo konstantnu i nekonstantnu verziju iste funkcije:

```
class Something
{
public:
    int m_nValue;

    const int& GetValue() const { return m_nValue; }
    int& GetValue() { return m_nValue; }
};
```

Konstantna verzija funkcije će biti pozvana samo kada se radi sa konstantnim objektima dok će nekonstantna funkcija biti pozvana pri radu sa nekonstantnim objektima:

```
Something cSomething;
cSomething.GetValue(); // calls non-const
GetValue();

const Something cSomething2;
cSomething2.GetValue(); // calls const GetValue();
```

Preklapanje funkcija sa konstantnom i nekonstantnom verzijom se praktikuje u slučaju kada povratna vrednost funkcije treba da se razlikuje kada je u pitanju konstantnost. U prethodnom primeru, konstantna verzija funkcije GetValue() vraća konstantnu referencu kao rezultat, dok će nekonstantni oblik funkcije GetValue() vratiti referencu koja nije deklarisana kao konstantna.

PRIMERI UPOTREBE KONSTANTNIH FUNKCIJA

Funkcije pristupa podacima klase koje služe samo za očitavanje podataka (geter funkcije), tj koje sigurno ne menjaju sadržaj, treba da budu definisane kao konstantne funkcije

Pogledajmo sledeći primer, gde ćemo u klasi za datum Date načiniti funkciju konstantnom kako bi mogla biti korišćenja nad konstantnim objektom klase Date. Početna verzija klase je:

```
class Date
{
private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;

    Date() { } // private default constructor

public:
    Date(int nMonth, int nDay, int nYear)
    {
        SetDate(nMonth, nDay, nYear);
    }
    void SetDate(int nMonth, int nDay, int nYear)
    {
        m_nMonth = nMonth;
        m_nDay = nDay;
        m_nYear = nYear;
    }
    int GetMonth() { return m_nMonth; }
    int GetDay()  { return m_nDay; }
    int GetYear() { return m_nYear; }
};
```

Jedina funkcija članica koja ne vrši promenu podataka klase (niti se iz nje pozivaju druge funkcije koje menjaju vrednosti podataka objekta) su funkcije pristupa. Stoga, ove funkcije treba da budu načinjene konstantnim. U nastavku je verzija koda gde su sve funkcije koje ne menjaju sadržaj objekata klase Date definisane kao konstantne:

```
class Date
{
private:
    int m_nMonth;
    int m_nDay;
    int m_nYear;

    Date() { } // private default constructor

public:
    Date(int nMonth, int nDay, int nYear)
    {
        SetDate(nMonth, nDay, nYear);
    }
    void SetDate(int nMonth, int nDay, int nYear)
    {
        m_nMonth = nMonth;
        m_nDay = nDay;
        m_nYear = nYear;
    }
    int GetMonth() const { return m_nMonth; }
    int GetDay() const { return m_nDay; }
    int GetYear() const { return m_nYear; }
};
```

Specifikatori pristupa članovima klase

učauravanje, specifikatori pristupa, public, private, protected

- *Tipovi specifikatora pristupa*
- *Javni članovi klase - public*
- *Privatni članovi klase - private*
- *Zaštićeni članovi klase - protected*

08

TIPOVI SPECIFIKATORA PRISTUPA

Ograničavanje pristupa članovima klase se specificira korišćenjem labela public, private i protected (tj. specifikatora pristupa) kojima se označavaju odgovarajuće sekcije unutar tela klase

Sakrivanje podataka je jedno od glavnih odlika objektno orijentisanog programiranja, koje obezbeđuje da se funkcijama programa (korisnicima klase) spreči direktan pristup unutrašnjim elementima klase. Ograničavanje pristupa članovima klase se specificira korišćenjem labela public, private, i protected kojima se označavaju odgovarajuće sekcije unutar tela klase. Ključne reči public, private, i protected se nazivaju specifikatorima pristupa.

Unutar jedne klase može da postoji više sekcija označenih labelama public, protected, ili private. Svaka sekcija je pod uticajem jednog specifikatora sve dok se ne navede nova labela ili dok se ne zatvori desna vitičasta zagrada koja predstavlja kraj definicije klase. Podrazumevajuću specifikator pristupa, kada se ne navede nijedan drugi, je private.

```
class Base {
   public:
  // public members go here
   protected:
  // protected members go here
   private:
  // private members go here
};
```

JAVNI ČLANOVI KLASE - PUBLIC

Javnim članovima klase - public – moguće je pristupiti iz bilo kog dela koga van klase, ali naravno u okviru programa

Javnim članovima klase - public – moguće je pristupiti iz bilo kog dela koga van klase, ali naravno u okviru programa. Kao što je navedeno u sledećem primeru moguće je dobiti informaciju o javnom članu ili ga promeniti bez korišćenja funkcija članica klase:

```
#include <iostream>
using namespace std;

class Line
{
   public:
      double length;
      void setLength( double len );
      double getLength( void );
};

// Member functions definitions
double Line::getLength(void)
{
    return length ;
}

void Line::setLength( double len )
{
    length = len;
}
```

Glavni program može biti napisan na sledeći način:

```
// Main function for the program
int main( )
{
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength()
<<endl;

   // set line length without member function
   line.length = 10.0; // OK: because length is
public
   cout << "Length of line : " << line.length
<<endl;
   return 0;
}
```

Rezultat prethodnog programa će biti:

```
Length of line : 6
Length of line : 10
```

PRIVATNI ČLANOVI KLASE - PRIVATE

Privatni članovi klase nisu vidljivi niti im se može direktno pristupiti izvan klase u kojoj su definisani. Podrazumeva se da su svi članovi klase privatni kada nije naveden specifikator

Privatni članovi klase (podaci i funkcije klase označeni kao private) nisu vidljivi niti im se može direktno pristupiti izvan klase u kojoj su definisani. Jedino funkcije članice klase, ili prijateljske funkcije (o kojima će biti više reči u sledećoj lekciji) mogu da pristupe i menjaju privatne članice klase.

Podrazumeva se da su svi članovi klase privatni kada nije naveden nijedan specifikator pristupa. Tako, na primer, u sledećoj klasi Box polje width je privatni podatak klase, što znači da svi članovi klase pre navođenja bilo koje labele su podrazumevano privatni (private):

```
class Box
{
          double width;
public:
          double length;
          void setWidth( double wid );
          double getWidth( void );
};
```

Praksa je da se podaci definišu u okviru sekcije označene kao private, a odgovarajuće povezane funkcije u javnoj (public) sekciji tako da te funkcije mogu biti pozvane van klase u cilju pristupa i promene privatnih članova, kao što je navedeno u sledećem primeru:

```
#include <iostream>

using namespace std;

class Box
{
   public:
      double length;
      void setWidth( double wid );
      double getWidth( void );

   private:
      double width;
};

// Member functions definitions
double Box::getWidth(void)
{
    return width ;
}

void Box::setWidth( double wid )
{
    width = wid;
}

// Main function for the program
int main( )
{
   Box box;
   // set box length without member function
   box.length = 10.0; // OK: because length is public
   cout << "Length of box : " << box.length <<endl;

   // set box width without member function
   // box.width = 10.0; // Error: because width is private
   box.setWidth(10.0);  // Use member function to set it.
   cout << "Width of box : " << box.getWidth() <<endl;

   return 0;
}
```

ZAŠTIĆENI ČLANOVI KLASE - PROTECTED

Zaštićeni članovi klase - protected – su veoma slični privatnim (private) članovima klase, ali imaju jednu dodatnu pogodnost a to je da im se može pristupiti iz nasleđenih klasa

Zaštićeni članovi klase - protected – su veoma slični privatnim (private) članovima klase, ali imaju jednu dodatnu pogodnost a to je da im se može pristupiti iz nasleđenih klasa, koje se drugačije nazivaju izvedene klase. O izvedenim klasama i nasleđivanju će biti više reči u narednoj lekciji. Za sada, dovoljno će biti da pogledamo sledeći primer gde smo kreirali klasu SmallBox čija je roditeljska klasa Box.

```
#include <iostream>
using namespace std;
class Box
{
   protected:
      double width;
};
class SmallBox:Box // SmallBox is the derived class.
{
   public:
      void setSmallWidth( double wid );
      double getSmallWidth( void );
};
// Member functions of child class
double SmallBox::getSmallWidth(void)
{
    return width ;
}
void SmallBox::setSmallWidth( double wid )
{
    width = wid;
}
```

Ovaj primer je sličan jednom prethodno navedenom primeru gde smo imali podatak width član klase Box. U ovom primeru smo definisali width kao zaštićen član kome je moguć pristup iz funkcija članica klase SmallBox, naslednice klase Box. Glavni program može biti napisan na sledeći način:

```
// Main function for the program
int main( )
{
   SmallBox box;

   // set box width using member function
   box.setSmallWidth(5.0);
   cout << "Width of box : "<< box.getSmallWidth()
<< endl;

   return 0;
}
```

Rezultat prethodnog programa biće:

```
Width of box : 5
```

Vežbe

klase, objekti, konstruktori, destruktori,specifikatori pristupa

- *Redosled poziva konstruktora i destruktora*
- *Studija slučaja – Kreiranje i upotreba klase Time*

09

PRIMER. PODACI I FUNKCIJE ČLANICE KLASE

Kreirati klasu Fakultet koja sadrži podatke: adresa, broj zaposlenih, zarada i funkciju članicu info() tipa void. Napisati glavni program u cilju testiranja funkcionalnosti klase

Na primeru koji sledi videće se sadržaj jednog jednostavnog i kompletnog C++ programa. U primeru je deklarisana klasa Fakultet koja sadrži podatke: adresa, broj zaposlenih, zarada i funkciju članicu info() tipa void. Svi članovi klase su smešteni u blok public što znači da se svima njima može pristupiti sa bilo kog mesta iz programa. Sam program se sastoji samo od funkcije main() u kojoj se dodeljuju vrednosti članovima deklarisanog objekta "fit" klase Fakultet. Pored toga, vrši se poziv funkcije članice info() koja ima zadatak da odštampa neke podatke koji pripadaju objektu kome pripada i sama ova funkcija.

```
#include <stdio.h>
#include <string.h>

class Fakultet
{
public:
            char adresa[30];
            int br_zaposlenih;
            float zarada;
            void info(void);
};

void Fakultet::info(void)
{
            printf("adresa:  %s\n",adresa);
            printf("broj zaposlenih:
%d\n",br_zaposlenih);
}

void main(void)
{
            Fakultet fit;

            fit.br_zaposlenih=150;
            strcpy(fit.adresa,"Tadeusa Koscuska br.
63");

            fit.info();
}
```

PRIMER. JAVNE I PRIVATNE ČLANICE KLASE

Premestiti podatak o zaradi iz prethodnog primera u privatnu sekciju klase Fakultet i kreirati odgovarajuću pristupnu funkciju koja će moći da očita sadržaj privatnog podatka

U uvodnom delu o OOP već je obrazložen koncept apstrakcije kojim se podaci i funkcije grupišu po osnovu "vidljivosti" za dalje korisnike programiranih klasa. Ne ulazeći u razloge zašto je neki od članova potrebno ili jednostavno zgodno proglasiti za nevidljive drugima, odnosno privatne za datu klasu, možemo u prethodnom primeru promenljivu zarada izdvojiti u sekciju private. Na ovaj način promenljivoj zarada se ne može direktno pristupiti iz programa, već samo uz pomoć neke funkcije članice klasa (funkcije članice imaju pristup privatnim promenljivama). Naravno, uslov je da ta funkcija bude javna - public. Tako je u primeru pridodata funkcija infoz, članica klase Fakultet, koja daje vrednost zaštićene promenljive.

```
class Fakultet
{
public:
            char adresa[30];
            int br_zaposlenih;
            void info(void);
            float infoz(void);
private:
            float zarada;
};
```

Kako ovoj promenljivoj nije dodeljena nikakva vrednost tj. nije inicijalizovana, očitana vrednost biće slučajna veličina zavisna od računara i kompajlera koji je upotrebljen

Da bi se iz programa (iz funkcije main) dodelila vrednost promenljivoj zarada, morala bi se, za tu svrhu, dodati posebna funkcija članica klase:

```
#include <stdio.h>
#include <string.h>

void Fakultet::info(void)
{
            printf("adresa:  %s\n",adresa);
            printf("broj zaposlenih:
%d\n",br_zaposlenih);
}
float Fakultet::infoz(void)
{
            return zarada;
}
void main(void)
{
            Fakultet fit;

            fit.br_zaposlenih=150;
            strcpy(fit.adresa,"Sestre Janjic br.
6");

            fit.info();

            // privatnoj promenljivoj ne moze se
direktno pristupiti
            //    fit.zarada=100000.;
            // javna funkcija klase moze da se
upotrebi za to
            printf("zarada:  %f\n",fit.infoz());
}
```

PRIMER. PODRAZUMEVAJUĆI I KONSTRUKTOR SA PARAMETRIMA

Kreirati podrazumevajući i konstruktor sa parametrima za klasu Rectangle, kao i metodu koja računa površinu odgovarajućeg geometrijskog tela (pravougaonika)

U nastavku je dat primer kontruktora klase koja služi za manipulacijom nad pravouganikom kao geometrijskim objektom:

```
// overloading class constructors
#include <iostream>
using namespace std;

class Rectangle
{
    int width, height;
  public:
    Rectangle ();
    Rectangle (int,int);
    int area (void) {return (width*height);}
};

Rectangle::Rectangle ()
{
  width = 5;
  height = 5;
}

Rectangle::Rectangle (int a, int b)
{
  width = a;
  height = b;
}
```

U primeru možemo primetiti da su kreirana dva konstruktora: podrazumevajući i konstruktor sa parametrima

U cilju testiranja kreiranih konstruktora napisaćemo sledeći program:

```
int main ()
{
  Rectangle rect (3,4);
  Rectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Konstruktor za ovu klasu može biti definisan, kao i obično, na sledeći način:

```
Rectangle::Rectangle (int x, int y) { width=x; height=y; }
```

Ali može biti definisan pomoću liste za inicijalizaciju:

```
Rectangle::Rectangle (int x, int y) : width(x) { height=y; }
```

Ili čak i na sledeći način:

```
Rectangle::Rectangle (int x, int y)
            : width(x), height(y) { }
```

Primetimo kako u ovom poslednjem slučaju da konstruktor vrši inicijalizaciju članova pomoću liste za incijalizaciju iako je telo konstrukotra prazno.

PRIMER. KONSTRUKTOR I DESTRUKTOR

Kreirati klasu Krug koja ima dinamički tekstualni podatak o boji kruga. Kreirati odgovarajući konstruktor i destruktor koji zauzima/oslobađa dinamički deo memorije za boju kruga

Iz do sada prikazanih primera vidi se da nije neophodno formirati konstruktor i destruktor za svaku klasu, već se to čini prema potrebama programera. Pisanje korisnički definisanih konstruktora i destruktora predstavlja preklapanje ovih funkcija koje se automatski, od strane kompajlera, pridružuju svakoj klasi. Neka je klasa Krug deklarisana na sledeći način:

```
class Krug
{
public:
            int x,y,r;
            char *boja;
            Krug(void);         // konstruktor
            ~Krug(void);        // destruktor
};
```

Konstruktor klase definišemo kao:

```
Krug::Krug(void)
{
            printf("(konstruktor)
inicijalizacija\n");
            x=10;
            y=10;
            r=5;
            boja=(char *) malloc(10);
            strcpy(boja,"bela");
}
```

a destruktor kao:

```
Krug::~Krug(void)
{
            printf("(destruktor) oslobadjanje
memorije\n");
            free(boja);
}
```

U cilju testiranja prethodne klase napisaćemo sledeći glavni program:

```
void main(void)
{
            Krug A;

            printf("\nKrug\n x=%d\n y=%d\n r=%d\n
boja=%s\n\n",
                        A.x, A.y, A.r, A.boja);

            printf("objekat je unisten\n");
}
```

U cilju da obezbedimo neophodnu funkcionalnost programu moramo uključiti u naš projekat sledeće standardne bibliotečne fajlove:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

PRIMER: DEFINISANJE KLASE U POSEBNOM FAJLU RADI BOLJE UPOTREBLJIVOSTI U DRUGIM KLASAMA

U nastakvu je dat primer definicije klase GradeBook, koja je obavljena u posebnom fajlu GradeBook.h, koji se kasnije može uključiti u fajl sa glavnim main programom

Definicija klase GradeBook ima sledeći oblik:

```
// GradeBook.h
// GradeBook class definition in a separate file from main.
#include <iostream>
using std::cout;
using std::endl;

#include <string> // class GradeBook uses C++ standard string
class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
                // constructor initializes courseName with string
supplied as argument
                GradeBook( string name )
                {
                                setCourseName( name ); // call set
function to initialize courseName
                } // end GradeBook constructor

                // function to set the course name
                void setCourseName( string name )
                {
                                courseName = name; // store the
course name in the object
                } // end function setCourseName

                // function to get the course name
                string getCourseName()
                {
                                return courseName; // return
object's courseName
 // display a welcome message to the GradeBook user
 void displayMessage()
 {
  // call getCourseName to get the courseName
  cout << "Welcome to the grade book for\n" << getCourseName()
    << "!" << endl;
 } // end function displayMessage
private:
 string courseName; // course name for this GradeBook
}; // end class GradeBook
```

U cilju testiranja klase class GradeBook, neophodno je napisati razdvojen fajl sa izvornim kodom koji će sadržati main funkciju koja instancira i koristi objekte ove klase. Primer:

```
// Including class GradeBook from file GradeBook.h
for use in main.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h" // include definition of
class GradeBook

int main()
{
   // create two GradeBook objects
   GradeBook gradeBook1( "CS101 Introduction to C++
Programming" );
   GradeBook gradeBook2( "CS102 Data Structures in
C++" );

   // display initial value of courseName for each
GradeBook
   cout << "gradeBook1 created for course: " <<
gradeBook1.getCourseName()
      << "\ngradeBook2 created for course: " <<
gradeBook2.getCourseName()
      << endl;
   return 0; // indicate successful termination
} // end main
```

PRIMER: KREIRANJE KLASA, OBJEKATA I FUNKCIJA

U nastavku je dat izvorni kod programa gde se kreiraju klase, objekti I funkcije članice, odnosno kreiraju i koriste konstruktori.

Potrebno je proučiti program i proveriti da li ima grešaka. Identifikovati precizno šta program radi. Fajl friend.h:

```
#include <string>
#include <iostream>
using namespace std;
class Friend
{
public:
            Friend(string, int);
            ~Friend();
```

Fajl friend.cpp:

```
void Friend::Display()
{
            cout << surname << phone << endl;
}
//constructor/desctructor
Friend::Friend(string initsurname, int initphone)
{
            phone = initphone;
```

Fajl main.cpp:

```
#include <iostream>
int main()
{
            //Friend Joe;
            Friend Joe("Smith", 1234567);
            Joe.Display();
            Joe.surname = "Smith";
            Joe.phone = 1234567;
```

Alternativni primer može biti napisan na sledeći način:

Fajl friend.h

```
//main.cpp:
#include "friend.h"
int main()
{
            Friend Joe("Smith", 1234567);
            Joe.Display();
            Joe.Assignphone(7654321);
            cout << Joe.phone << endl; return 0;
}
```

Fajl friend.h

```
//friend.h:
#include <string>
#include <iostream>
using namespace std;
class Friend
{
public:
            Friend(string initsurname, int initphone)
            {
                        phone = initphone;
                        surname = initsurname;
            }
            ~Friend() { }
            void Display()
            {
                        cout << surname << phone << endl;
 }
 void Assignphone(int newphone)
 {
  phone = newphone;
 }
 string surname;
 int phone;
};
```

Redosled poziva konstruktora i destruktora

konstruktor i destruktor, global, local, static

- *Definisanje konstruktora i destruktora klase CreateAndDestroy*
- *Kada se pozivaju konstruktor i destruktor?*

09

DEFINISANJE KONSTRUKTORA I DESTRUKTORA KLASE CREATEANDDESTROY

Kreirati proizvoljnu klasu, i za datu klasu kreirati globalne, lokalne i statičke instance. Ispitati redosled poziva konstruktora i destruktora svakog kreiranog objekta

Program koji sledi demonstrira redosled u kome se pozivaju konstruktor i destruktor za objekte klase CreateAndDestroy različitih memorijskih klasa u nekoliko različitih opsega promenljivih. Svaki od objekata klase sadrži ceo broj (objectID) i string (message) koji se zatim štampaju na standardnom izlazu da bi se identifikovalo o kom se objektu radi. Deklaracija klase CreateAndDestroy je napisana na sledeći način.

```
// CreateAndDestroy.h
// Definition of class CreateAndDestroy.
// Member functions defined in CreateAndDestroy.cpp.
#include <string>
using std::string;

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy
{
public:
   CreateAndDestroy( int, string ); // constructor
   ~CreateAndDestroy(); // destructor
private:
   int objectID; // ID number for object
   string message; // message describing object
}; // end class CreateAndDestroy

#endif
```

Ovakav mehanički primer je kreiran iz čisto pedagoških razloga. U ovu svrhu, u okviru destruktora imamo liniju koja ispituje da li objekat koji se uništava, ima vrednost podatka objectID 1 ili 6 i, ukoliko je tako, štampa se karakter nove linije, Ova linija omogućava lakše praćenje onoga što se štampa na izlazu. Definicije funkcija klase CreateAndDestroy su smeštene u posebnom fajlu.

```
// CreateAndDestroy.cpp
// Member-function definitions for class CreateAndDestroy.
#include <iostream>
using std::cout;
using std::endl;

#include "CreateAndDestroy.h"// include CreateAndDestroy class
definition

// constructor
CreateAndDestroy::CreateAndDestroy( int ID, string messageString
)
{
   objectID = ID; // set object's ID number
   message = messageString; // set object's descriptive message

   cout << "Object " << objectID << "   constructor runs   "
      << message << endl;
} // end CreateAndDestroy constructor

// destructor
CreateAndDestroy::~CreateAndDestroy()
{
   // output newline for certain objects; helps readability
   cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );

   cout << "Object " << objectID << "   destructor runs    "
      << message << endl;
} // end ~CreateAndDestroy destructor
```

KADA SE POZIVAJU KONSTRUKTOR I DESTRUKTOR?

U C++ programu se prvo vrši uništavanje (destrukcija) lokalnih (u suprotnom redosledu od redosleda kreiranja), zatim statičkih, a tek na kraju globalno definisanih objekata

U kodu koji sledi vrši se testiranje definisane funkcionalnosti klase CreateAndDestroy.

```
// main.cpp
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>
using std::cout;
using std::endl;

#include "CreateAndDestroy.h" // include CreateAndDestroy class
definition

void create( void ); // prototype

CreateAndDestroy first( 1, "(global before main)" ); // global object

int main()
{
   cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy second( 2, "(local automatic in main)" );
   static CreateAndDestroy third( 3, "(local static in main)" );

   create(); // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
   CreateAndDestroy fourth( 4, "(local automatic in main)" );
   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
   return 0;
} // end main

// function to create objects
void create( void )
{
   cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy fifth( 5, "(local automatic in create)" );
   static CreateAndDestroy sixth( 6, "(local static in create)" );
   CreateAndDestroy seventh( 7, "(local automatic in create)" );
   cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
} // end function create
```

Prvo se definiše globalna instanca klase CreateAndDestroy, pod nazivom first. Njen konstruktor se ustvari poziva pre nego se izvrši bilo koja instrukcija glavne main funkcije, dok se njen destruktor poziva u trenutku prekida rada programa, tj. nakon poziva destruktora svih ostalih objekata definisanih u programu. U funkciji main se deklarišu tri objekta. Objekti second i fourth su lokalni auto objekti, a objekat third je statički (static) lokalni objekat. Konstuktor svakog od ovih objekata se poziva nakon što izvršenje programa dostigne liniju deklaracije ovih objekata. Destruktor objekta fourth i second se poziva (u obrnutom redosledu u odnosu na redosled poziva konstruktora) kada izvršenje programa dostigne liniju kraja funkcije main. S obzirom da je objekat third tipa static, on će živeti do prekida programa. Destruktor objekta third se poziva pre destruktora globalno objekta, first, a nakon poziva destruktora svih ostalih objekata. U funkciji create se deklarišu tri objekta: fifth i seventh kao lokalni automatski objekti, a sixth kao lokalni static objekat. Destruktor objekta seventh a zatim i objekta fifth se poziva (u obrnutom redosledu u odnosu na redosled poziva konstruktora) po izvršenju funkcije create. Obzirom da je sixth definisan kao static, on će živeti dok se ne izvrši program. Destruktor objekta sixth se poziva pre destruktora objekata third i first, ali nakon uništavanja ostalih objekata.

Studija slučaja – Kreiranje i upotreba klase Time

klasa Time, definicija klase, upotreba klase

- *Deklaracija klase Time*
- *Definisanje funkcija članica klase Time*
- *Definisanje funkcija članica klase Time*
- *Korišćenje klase Time*

09

DEKLARACIJA KLASE TIME

Kreirati klasu Time koja će da simulira rad časovnika

U ovom primeru biće kreirana klasa Time kao i pokretački program koji će da testira funkcionalnost definisane klase. U okviru primera biće obnovljeno puno koncepata koje smo pomenuli u prethodnom delu predavanja i vežbi. Takođe će biti spomenut deo naredbi pretprocesora koje se koriste sa ciljem da se neki fajl zaglavlja tokom kompajliranja ne uključi u projekat više od jedanput.

```
// Time.h
// Declaration of class Time.
// Member functions are defined in Time.cpp

// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H

// Time class definition
class Time
{
public:
   Time(); // constructor
   void setTime( int, int, int ); // set hour,
minute and second
   void printUniversal(); // print time in
universal-time format
   void printStandard(); // print time in standard-
time format
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 - 59
   int second; // 0 - 59
}; // end class Time
```

Definicija prethodne klase sastoji se iz prototipova funkcija članica klase Time: setTime, printUniversal i printStandard. Klasa sadrži privatne (private) podatke članove, kojima možemo pristupiti samo preko ove 4 definisane funkcije. Možemo primetiti da je definicija klase izvršena izmedju linija pretprocesorskih naredbi uslovnog prevođenja:

```
// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H
   ...
#endif
```

Kada vršimo kreiranje velikih programa, javiće se slučaj da imamo puno klasa deklarisanih u drugim fajlovima sa zaglavljima (header files). Prethodna pretprocesorska direktiva uslovnog prevođenja sprečava da se deo koda između #ifndef (što znači "ako nije definisano onda...") i #endif uključi u kod ukoliko je ime TIME_H već definisano. Ukoliko ovaj fajl zaglavlja nije prethodno uključen u neki fajl (pomoću #include „Time.h“) , ime TIME_H će biti definisano pomoću direktive #define i izvršiće se uključivanje ovog fajla zaglavlja. U suprotnom, ukoliko je ovaj fajl već uključen negde drugde, to znači da je i ime TIME_H već definisano pa će se sprečiti ponovno uključivanje ovog fajla Time.h.

DEFINISANJE FUNKCIJA ČLANICA KLASE TIME

U nastavku je dat fajl Time.Cpp u kome su definisane funkcije članice klase Time

Definicije funkcija članica klase Time ćemo smestiti u fajl Time.cpp. Izgled ovog fajla je dat u nastavku

```
// Time.cpp
// Member-function definitions for class Time.
#include <iostream>
using std::cout;

#include <iomanip>
using std::setfill;
using std::setw;

#include "Time.h" // include definition of class Time from
Time.h

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time()
{
   hour = minute = second = 0;
} // end Time constructor

// set new Time value using universal time; ensure that
// the data remains consistent by setting invalid values to
zero
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate
minute
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate
second
} // end function setTime
```

```
// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour <<
":"
      << setw( 2 ) << minute << ":" << setw( 2 ) <<
second;
} // end function printUniversal


// print Time in standard-time format (HH:MM:SS AM
or PM)
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour
% 12 ) << ":"
      << setfill( '0' ) << setw( 2 ) << minute <<
":" << setw( 2 )
      << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

DEFINISANJE FUNKCIJA ČLANICA KLASE TIME

U nastavku je dat fajl Time.Cpp u kome su definisane funkcije članice klase Time

U ovom kodu možemo primeti podrazumevajući konstruktor koji inicijalizuje sve podatke klase sa nulom (tj. 0 predstavlja vremenski trenutak ekvivalentan sa 12 AM). Ovo obezbeđuje da vrednost objekata počinje iz konzistentnog vremenskog trenutka. Pogrešne vrednosti ne mogu biti smeštene u polja objekta klase Time jer se konstruktor poziva pri kreiranju objekta, dok će svaki naredni pokušaj promene polja korišćenjem funkcije setTime biti uslovljen odgovarajućim pravilnim unosom (što se može primetiti u kodu). Naravno, programer može kreirati različiti broj konstruktora sa parametrima, koliko je to neophodno za pravilno funkcionisanje programa.

Članovi klase ne mogu biti inicijalizovani u telu klase, gde se vrši deklaracija. Strogo je preporučljivo da se ova polja objekta inicijalizuju korišćenjem konstruktora (pošto ne postoji podrazumevajuća inicijalizacija za ugrađene tipove podataka). Dodela vrednosti poljima klase može biti izvršena i korišćenjem funkcija setera.

KORIŠĆENJE KLASE TIME

U nastavku je dat prikaz fajla main.cpp u kome se nalazi main program u cilju testiranja kreirane funkcionalnosti klase Time

Nakon što definišemo neku klasu, kao u ovom slučaju klasu Time, ona može biti korišćena kao nezavisan tip podatka pri deklaraciji objekata, nizova, pokazivača i referenci, kao što je prikazano u nastavku:

```
Time sunset; // object of type Time
Time arrayOfTimes[ 5 ], // array of 5 Time objects
Time &dinnerTime = sunset; // reference to a Time object
Time *timePtr = &dinnerTime, // pointer to a Time object
```

U sledećem primeru je pokazan fajl sa pokretačkom main funkcijom koja koristi klasu. U prvoj liniji main funkcije se instancira jedan objekat klase Time pod nazivom t. Nakon instanciranja objekta poziva se konstruktor da inicijalizuje sa nulom sve privatne članove klase. Zatim se pozivaju funkcije koje štampaju vreme u univerzalnom i standardnom formatu. Zatim se u liniji:

```
t.setTime( 99, 99, 99 ); // attempt invalid settings
```

pravi pokušaj da se dodele nedozvoljene vrednosti ali funkcija setTime ovo prepoznaje i umesto nedozvoljenih vrednosti postavlja 0. Izgled main.cpp fajla je:

```
// main.cpp
// Program to test class Time.
// NOTE: This file must be compiled with Time.cpp.
#include <iostream>
using std::cout;
using std::endl;

#include "Time.h" // include definition of class Time from Time.h

int main()
{
   Time t; // instantiate object t of class Time

   // output Time object t's initial values
   cout << "The initial universal time is ";
   t.printUniversal(); // 00:00:00
   cout << "\nThe initial standard time is ";
   t.printStandard(); // 12:00:00 AM

   t.setTime( 13, 27, 6 ); // change time
 // output t's values after specifying invalid values
   cout << "\n\nAfter attempting invalid settings:"
      << "\nUniversal time: ";
   t.printUniversal(); // 00:00:00
   cout << "\nStandard time: ";
   t.printStandard(); // 12:00:00 AM
   cout << endl;
   return 0;
} // end main
```

Rezultat programa biće:

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
```

Zadaci za samostalan rad

klase, objekti, konstruktori, destruktori, specifikatori pristupa

- *Zadaci za samostalno vežbanje*
- *Zadaci za samostalno vežbanje - Dodatak*

10

ZADACI ZA SAMOSTALNO VEŽBANJE

Korišćenjem materijala sa predavanja i vežbi za ovu nedelju uraditi sledeće zadatke:

1. Napraviti klasu Helikopter koja u sebi sadrži sledeće podatke: registarskiBroj, maksimalnaBrzina, brojSedista. U pokretačkoj main funkciji napuniti datu klasu sa podacima i prikazati u liniji sadržaj klase.

2. U luci su usidreni čamci i jahte. Napraviti program za izračunavanje procenta čamaca ako kao parametar imamo ukupan broj plovila kao i broj jahti. Prilikom pravljenja ove aplikacije treba kreirati klasu koja predstavlja luku (i sve što se tiče luke) i main funkciju koja je zadužena za sakupljanje podataka i za prikaz podataka.

3. Napraviti klasu Kontakt koja sadrži atribute:

• ime

• brojTelefona

Pored standardnih setera i getera, dodati i metodu toString. Prikazati rad klase u main-u.

4. Napraviti klasu Student koja opisuje jednog studenta i pokretačku funkciju main koja demonstrira sve funkcionalnosti klase Student. Definisati klasu Student i kreirati sledeće atribute:

- ime, tipa String

- brojIndeksa, tipa int

Kreirati podrazumevani (ili prazan) konstruktor koristeći ključnu reč this, tj. konstruktor koji ne prima argumente, tako da postavlja ime na vrednost "Student X", a broj indeksa na vrednost 3000. Kreirati konstruktor sa parametrima koji prima vrednosti svih atributa. Kreirati konstruktor kopiranja, tj. konstruktor koji prima objekat klase Student kao argument. Kreirati metodu ispisi(), koja vraća vrednosti svih atributa u sledećem formatu:
Ime studenta: Student X
Broj indeksa studenta: 3000

ZADACI ZA SAMOSTALNO VEŽBANJE - DODATAK

U nastavku su date postavke dodatnih zadataka za samostalni rad:

5. Na osnovu klase Student kreirati klasu Profesor , tako da sadrže sve elemente koje ima klasa Student. Svaki profesor je opisan imenom, prezimenom, zvanjem i brojem godina. U main funkciji demonstrirati funkcionalnost kreirane klase Profesor.

6. Napraviti klasu Kvadrat koja opisuje jedan kvadrat i pokretačku funkciju main() koja demonstrira sve funkcionalnosti klase Kvadrat. Definisati klasu Kvadrat i kreirati sledeće javne atribute:
stranica, tipa int
bojaIvice, tipa String
bojaUnutrašnjosti, tipa String

Kreirati podrazumevani (iliti prazan) konstruktor koristeći ključnu reč this, tj. konstruktor koji ne prima argumente tako da postavlja podrazumevanu veličinu stranice na 1 i podrazumevane boje na vrednost "Crna". Kreirati metodu ispiši(), koja ispisuje vrednosti svih atributa u sledećem formatu, sa tabulatorima:
Stranica kvadrata: 1
Boja ivice kvadrata:Crna
Boja unutrašnjosti kvadrata:Crna

Koristeći pokretačku funkciju main demonstrirati sve trenutne funkcionalnosti klase Kvadrat. Kreirati sledeće metode koristeći ključnu reč this:
- računajPovršinu, koja vraća površinu kvadrata
- računajObim, koja vraća obim kvadrata.

Zaključak

12

O KLASAMA, OBJEKTIMA, ČLANICAMA KLASE, KONSTRUKTORIMA I DESTRUKTORIMA

Na osnovu svega obrađenog možemo zaključiti sledeće

Osnovna svrha C++ programiranja je da se doda objektna orijentacija C programskom jeziku. Klase su centralna odlika C++-a koja upravo podržava objektno orijentisano programiranje.

U cilju kreiranja objekta potrebno je kreirati „instancu“ tj primerak klase. Ovo se postiže na potpuno isti način kao i kreiranje primeraka tj. „instanci“ običnih tipova podataka.

Definicija klase počinje ključnom rečju class za kojim sledi ime klase i telo klase oivičeno parom vitičastih zagrada.

Deklaracija objekta se vrši na isti način kako se deklarišu promenljive osnovnih tipova podatka.

Javnim (public) podacima objekata klase može se pristupiti korišćenjem operatora pristupa (.), što nije dozvoljeno kod privatnih (private) i zaštićenih (protected) članova klase.

Funkcije članice klase su one funkcije čija se definicija ili prototip nalazi u okviru definicije (tela) klase. One mogu da operišu sa svim objektima klase čiji su član.

Najčešća praksa je da se deklaracija funkcije članice navede u okviru definicije klase, a da se implementacija izvrši van tela klase i to u nekom drugom fajlu.

Konstruktor je funkcija članica koja ima isto ime kao i klasa. Poziva se pri kreiranju objekta, dok se pri uništenju objekta poziva takođe funkcija istoga naziva kao i klasa koja se zove destruktor.

Konstruktor sa parametrima omogućava da se kroz listu argumenata konstruktora dodele inicijalne vrednosti članovima objekta u samom trenutku kreiranja objekta.

Destruktori su funkcije koje se, kao i konstruktori, pozivaju automatski, ali u ovom slučaju pri prestanku postojanja objekta tj. pri njegovom destruktuiranju.

Konstruktor kopiranja je konstruktor koji kreira objekat i inicijalizuje ga vrednostima objekta iste klase, koji je prethodno kreiran.

O POKAZIVAČU THIS, OBLASTI VAŽENJA KLASE, KONSTANTNIM ČLANOVIMA KLASE I SPECIFIKATORIMA PRISTUPA

Možemo izvesti sledeći zaključak:

Svaki objekat u C++-u ima pristup svojoj adresi (adresi na kojoj je smešten memoriji) pomoću jednog veoma bitnog pokazivača koji se naziva pokazivač this.

Odvajanjem definicije klase u header fajl veličina glavnog programa se smanjuje i klasa postaje modularna pa se može koristiti unutar bilo kojeg programa pomoću #include direktive.

U slučaju da drugi programer želi da dotičnu odvojenu klasu uključi u svoj program, treba da zna imena metoda klase i listu argumenata.

Pri pisanju složenog projekta preporuka je da deklaracija klase bude u header fajlu, definicija funkcija u .cpp fajlu istog naziva, a da se u fajl sa glavnim programom uključi header fajl klase.

Podaci članovi klase su automatski u području važenja funkcija članova klase. Stoga podatke članove nije potrebno navoditi kao parametre ili kao povratne vrednosti funkcija članova.

Jednom kad je konstantan objekat klase inicijalizovan pomoću konstruktora, bilo kakav pokušaj promene sadržaja obekta biće nedozvoljena operacija.

Konstantna funkcija članica je ona koja garantuje da neće promeniti vrednosti bilo kog podatka klasa, niti da će se iz nje pozvati funkcije koje nisu deklarisane kao konstantne.

Svaka konstantna funkcija članica koja pokuša da promeni vrednost podatka klase ili da pozove neku drugu funkciju članicu koja nije konstantna će da proizvede kompajlersku grešku.

Ograničavanje pristupa članovima klase se specificira korišćenjem labela public, private i protected (tj. specifikatora pristupa) kojima se označavaju odgovarajuće sekcije unutar tela klase.