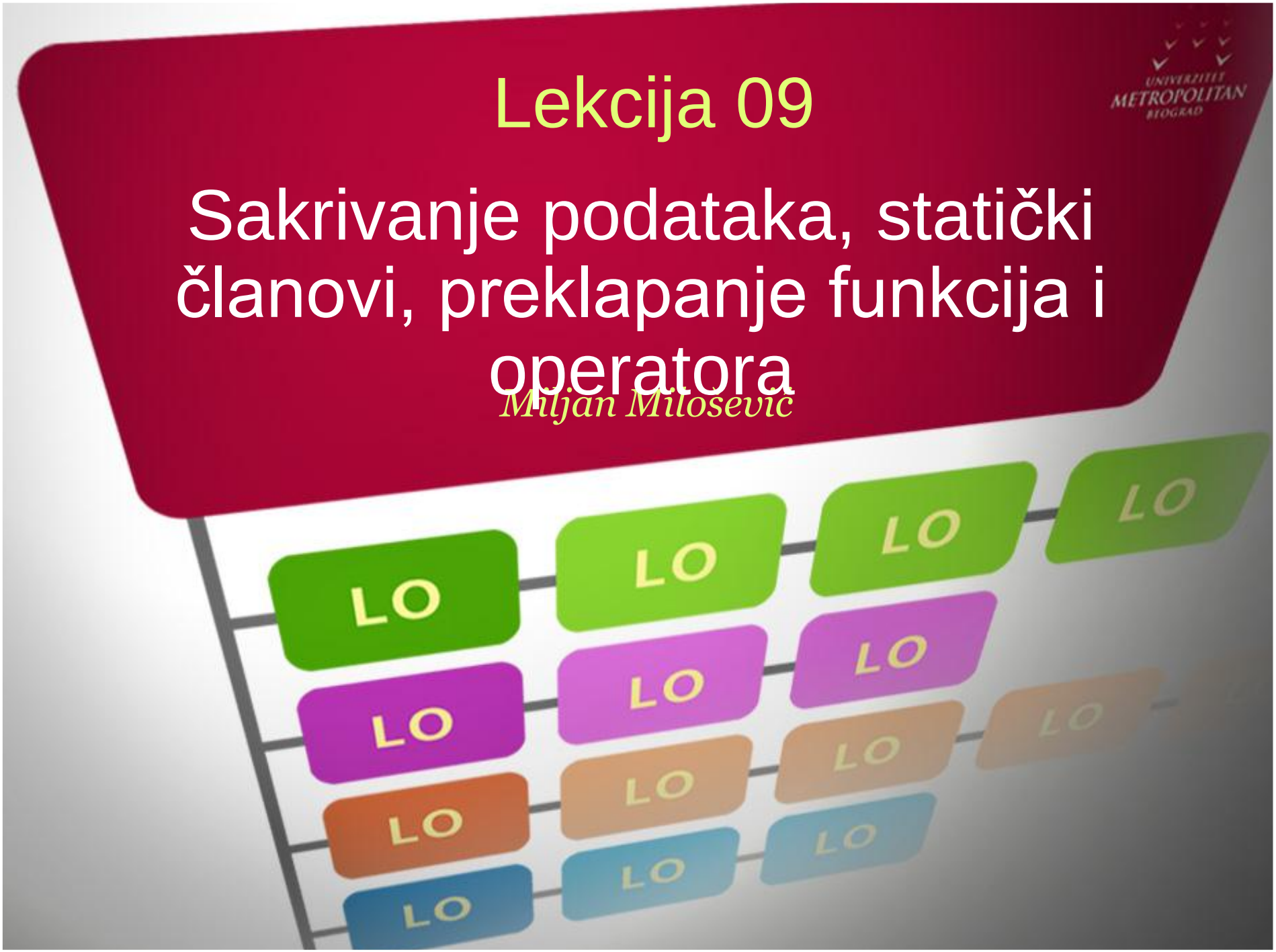
Lekcija 09

Sakrivanje podataka, statički članovi, preklapanje funkcija i operatora

Miljan Milošević



SAKRIVANJE PODATAKA, STATIČKI ČLANOVI, PREKLAPANJE FUNKCIJA I OPERATORA

Uvod

01 Učauravanje i sakrivanje podataka

- *Učauravanje - Enkapsulacija*
- *Primer enkapsulacije podataka*
- *Sakrivanje podataka – Funkcije pristupa*
- *Funkcije pristupa – Primer upotrebe*
- *Javni interfejs; Lokalne funkcije*

02 Prijateljske funkcije i klase

- *Uvod u prijateljske funkcije i klase*
- *Upotreba prijateljskih funkcija*
- *Prijateljske klase*

03 Statički članovi klase

- *Statički podaci klase*
- *Rad sa statičkim podacima objekata*
- *Statičke funkcije članice klase*
- *Upotreba statičkih podataka i funkcija klase*

04 Anonimne promenljive i objekti

- *Uvodna razmatranja*
- *Anonimni objekti klase*
- *Upotreba anonimnih objekata*

SAKRIVANJE PODATAKA, STATIČKI ČLANOVI, PREKLAPANJE FUNKCIJA I OPERATORA

05

Uvod u preklapanje funkcija i operatora

- *Preklapanje funkcija i operatora*
- *Preklapanje funkcija*
- *Preklapanje operatora (operator overloading)*
- *Primeri preklapanja operatora*

06

Preklapanje unarnih i binarnih operatora

- *Preklapanje operatora umanjenja ++ i uvećanja --*
- *Preklapanje unarnog operatora (-)*
- *Preklapanje binarnih operatora*
- *Primer preklapanja operatora sabiranja +*
- *Preklapanje operatora korišćenjem prijateljskih funkcija*
- *Preklapanje*

07

Preklapanje operatora poređenja i dodele

- *Preklapanje operatora poređenja*
- *Preklapanje operatora dodele vrednosti =*

08

Preklapanje ulazno/izlaznih operatora

- *Osnovi o preklapanju ulazno/izlaznih operatora*
- *Primer preklapanja ulazno/izlaznih operatora*

09

Preklapanje operatora poziva funkcije ()

- *Osnovi o preklapanju operatora poziva funkcije ()*

SAKRIVANJE PODATAKA, STATIČKI ČLANOVI, PREKLAPANJE FUNKCIJA I OPERATORA

10

Preklapanje operatora indeksiranja []

- *Postupak preklapanja operatora indeksiranja []*
- *Primena preklapanja operatora indeksiranja []*

11

Preklapanje operatora konverzije tipa

- *Uvodna razmatranja*
- *Preklapanje operacije konverzije tipa*
- *Primena preklapanja konverzije tipa*

12

Vežbe – Skrivanje podataka, Statički članovi klase

- *Studija slučaja - Klasa Time – Pristupne funkcije*
- *Primer - Statički članovi klase*

13

Vežbe – Preklapanje operatora

- *Studija slučaja – Klasa Date*

14

Zadaci za samostalan rad

- *Zadaci za samostalno vežbanje*

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

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

- Učauravanje i sakrivanje podataka
- Prijateljske funkcije i klase
- Statički članovi klase
- Anonimne promenljive i objekti
- Preklapanje funkcija i operatora

Učauravanje (enkapsulacija) je koncept objektno-orijentisanog programiranja koji obezbeđuje mogućnost da se kreira model realnog objekta u kome su sjedinjeni njegova svojstva i ponašanje. C++ obezbeđuje korisnicima klase osobine enkapsulacije i skrivanja podataka kroz kreiranje korisnički definisanog tipa, koji se naziva klasa. Već smo spomenuli da klasa može da sadrži privatne (private), zaštićene (protected) i javne (public) članove.

Kod kreiranja klasa, često se sve promenljive tj. podaci postave u privatni deo klase, a metode za pristup podacima se postavljaju u javni deo klase. Ovaj pristup, označen sa "javni interfejs a privatni podaci" (public interface, private data), je dakle glavni tj. osnovni koncept koji se koristi kod kreiranja klasa.

Ponekad, neki isti podatak je potreban za sve članove klase. Nepotrebno je i neefikasno bi bilo da se ovakva neka vrednost (podatak) memoriše u svakom objektu ponaosob. Zato, takav podatak se može proglasiti „statičkim" tako što se stavi ključna reč *static* na početku deklaracije nekog polja klase.

C++ dozvoljava postojanje prijateljskih funkcija klase koje su definisane van opsega klase ali imaju prava pristupa svim privatnim (private) i zaštićenim (protected) članovima klase. Osim prijateljskih funkcija, neka klasa takođe može biti prijateljska drugoj klasi.

Takođe, C++ dozvoljava definisanje više od jedne funkcije istog naziva kao i više operatora, što se naziva **preklapanje funkcija** odnosno **preklapanje operatora**, respektivno. Osim termina preklapanje često se koristi i termin predefinisanje (overloading).

Učauravanje i sakrivanje podataka

učauravanje, sakrivanje podataka, funkcije pristupa

- *Učauravanje - Enkapsulacija*
- *Primer enkapsulacije podataka*
- *Sakrivanje podataka – Funkcije pristupa*
- *Funkcije pristupa – Primer upotrebe*
- *Javni interfejs; Lokalne funkcije*

01

UČAURAVANJE - ENKAPSULACIJA

Enkapsulacijom se korisnicima klase ograničava direktan pristup njenim skrivenim delovima, da bi se smanjila mogućnost da se poljima objekata dodele pogrešne vrednosti

Svi C++ programi se sastoje iz sledećih osnovnih elemenata:

- **Ponašanja programa:** Ovo je deo programa koji obavlja odgovarajuće akcije i taj deo je predstavljen funkcijama.
- **Podaci programa:** Podaci su informacije programa koji se menjaju pod uticajem funkcija programa.

Učauravanje (enkapsulacija) je koncept objektno-orijentisanog programiranja koji obezbeđuje mogućnost da se kreira model realnog objekta u kome su sjedinjeni njegova svojstva i ponašanje. Enkapsulacijom se korisnicima klase ograničava direktan pristup njenim skrivenim delovima, da bi se smanjila mogućnost da se poljima objekata dodele pogrešne vrednosti. Ovo je jedan od načina kojima se obezbeđuje pouzdanost programa. Učauravanje podataka vodi do veoma važnog koncepta OOP koji se naziva sakrivanje podataka.

C++ obezbeđuje korisnicima klase osobine enkapsulacije i skrivanja podataka kroz kreiranje korisnički definisanog tipa, koji se naziva klasa. Već smo spomenuli da klasa može da sadrži privatne (private), zaštićene (protected) i javne (public) članove. Podrazumevano, u C++-u, svi članovi (polja) klase se tretiraju kao privatni. Pogledajmo sledeći primer:

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

Promenljive length, breadth, i height su deklarisane kao private. Ovo znači da im se može pristupiti samo pomoću metoda klase Box, dok im ostale metode programa, koji su korisnici klase, ne mogu prostupiti. Na ovaj način je izvršen jedan oblik enkapsulacije podataka.
Kao što smo napomenuli, da bi neke delove klase proglasili za javne (tj, dostupne u ostalim delovima programa), moramo ih deklarisati kao public. Sva polja ili funkcije definisane nakon ključne reči public su vidljive iz svih delova programa van klase. Idealan scenario je da se što je moguće više detalja jedne klase drži sakriveno od ostalih klasa.

PRIMER ENKAPSULACIJE PODATAKA

Svaki C++ program u kome je implementirana klasa sa privatnim članovima i javnim funkcijama preko kojih se pristupa privatnim članovima je jedan primer enkapsulacije podataka

Bilo koji primer C++ programa u kome je implementirana klasa sa javnim i privatnim članovima je jedan primer enkapsulacije podataka. Posmatrajmo sledeći primer:

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

class Adder
{
   public:
      // constructor
      Adder(int i = 0)
      {
        total = i;
      }
      // interface to outside world
      void addNum(int number)
      {
          total += number;
      }
      // interface to outside world
      int getTotal()
      {
          return total;
      };
   private:
      // hidden data from outside world
      int total;
};
```

Neka je glavni program napisan na sledeći način:

```
int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

Rezultat će biti:

```
Total 60
```

Prethodna klasa vrši dodavanje brojeva na već postojeću vrednost i na kraju, kao razultat, glavnom programu vraća ukupnu sumu. Javne funkcije članice addNum i getTotal su interfejsi dostupni spoljašnjem svetu, i korisnik jedino treba da poznaje da one postoje kako bi uspešno iskoristio mogućnosti klase. Privatni podatak total je nešto što je sakriveno od spoljašnjeg sveta, ali je neophodan da bi klasa imala osnovnu funkcionalnost.

SAKRIVANJE PODATAKA – FUNKCIJE PRISTUPA

Funkcije pristupa su kratke funkcije članice klase, označene kao javne (public), čija je svrha da pristupe vrednosti privatnog člana klase

Da bi se podaci zaštitili, članovi klase se mogu organizovati tako da ne budu direktno pristupni izvan klase. Ovo se zove 'sakrivanje podataka', tj data hiding.

Funkcije pristupa (access functions)

Funkcije pristupa su kratke funkcije članice klase, označene kao javne (public), čija je svrha da pristupe vrednosti privatnog člana klase. Na primer, u sledećoj klasi String, imamo:

```
class String
{
private:
    char *m_chString; // a dynamically allocated
string
    int m_nLength; // the length of m_chString

public:
    int GetLength() { return m_nLength; }
};
```

gde je GetLength() funkcija pristupa koja kao rezultat vraća vrednost privatne promenljive m_nLength.
Funkcije pristupa dolaze u dva oblika: geter i seter. Geteri (očitati vrednost – engl. getters) su funkcije koje jednostavno vraćaju vrednost privatnog podatka klase. Seteri (postaviti vrednost - setters) su funkcije pomoću kojih se menja vrednost privatnih članova klase.

Seter/geter funkcije se definišu na uobičajeni način, s tim da se imenuju po promenljivama kojima pristupaju. Tu se primenjuje konvencija da pri kreiranju naziva funkcije uključimo naziv promenljive, s tim što prvo slovo postaje veliko a u ime funkcije se ubaci na početku 'set' ili 'get', respektivno. U nastavku je dat primer definicije klase sa geter i seter funkcijama:

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

public:
    // Getters
    int GetMonth() { return m_nMonth; }
    int GetDay() { return m_nDay; }
    int GetYear() { return m_nYear; }

    // Setters
    void SetMonth(int nMonth) { m_nMonth = nMonth; }
    void SetDay(int nDay) { m_nDay = nDay; }
    void SetYear(int nYear) { m_nYear = nYear; }
};
```

FUNKCIJE PRISTUPA – PRIMER UPOTREBE

Pri kreiranju naziva funkcije pristupa obično se uključi naziv promenljive, s tim što prvo slovo postaje veliko a u ime funkcije se ubaci na početku 'set' ili 'get', respektivno

U nastavku je dat primer sakrivanja podataka u klasi Pas, gde smo uključili konstruktor u cilju da elegantno, jednom instrukcijom, inicijalizujemo objekat:

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

class Pas
{
public:
                Pas(int inicStarost, int inicTezina, string inicBoja);
                ~Pas();
                void setStarost(int starost);
                void setTezina(int tezina);
                void setBoja(string boja);
                int getStarost();
                int getTezina();
                string getBoja();
                void lajati();
private:
                int starost;
                int tezina;
                string boja;
};

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

Pas::~Pas(){}

void Pas::setStarost(int godine)
{
 starost = godine;
}

void Pas::setTezina(int kilogrami)
{
 tezina = kilogrami;
}
void Pas::setBoja(string novaBoja)
{
 boja = novaBoja;
}
```

```
int Pas::getStarost()
{
 return starost;
}
int Pas::getTezina()
{
 return tezina;
}
string Pas::getBoja()
{
 return boja;
}
void Pas::lajati()
{
 cout << "Av,av" << endl;
```

Glavni program možemo da napišemo na dva načina. Prvi je:

```
int main()
{
            Pas Jimi(3, 15, "crna");
            cout << Jimi.getStarost() << endl;
            cout << Jimi.getTezina() << endl;
            cout << Jimi.getBoja() << endl;
            return 0;
}
```

a drugi je bez upotrebe konstruktora:

```
int main()
{
            Pas Jimi;
            Jimi.setStarost(3);
            Jimi.setTezina(15);
            Jimi.setBoja("crna");
            cout << Jimi.getStarost() << endl;
            cout << Jimi.getTezina() << endl;
            cout << Jimi.getBoja() << endl;
            return 0;
}
```

JAVNI INTERFEJS; LOKALNE FUNKCIJE

„Javni interfejs a privatni podaci“ je osnovni koncept koji se koristi kod kreiranja klasa. Lokalne funkcije su privatne funkcije, vidljive samo u okviru drugih javnih funkcija članica klase

- JAVNI INTERFEJS, PRIVATNI PODACI:

Kod kreiranja klasa, često se sve promenljive tj. podaci postave u privatni deo klase, a metode za pristup podacima se postavljaju u javni deo klase. Ovaj pristup, označen sa “javni interfejs a privatni podaci” (public interface, private data), je dakle glavni tj. osnovni koncept koji se koristi kod kreiranja klasa. Prototipovi funkcija zadavanja i pristupa sakrivenim podacima se dodaju javnom delu klase, u cilju da se omogući zadavanje i učitavanje privatnih podataka. U nastavku je dat primer:

```
class Friend
{
public:
            void setAge(int years); int getAge();
            void setPhone(int number); int
getPhone();
            void setSurname(string yoursurname);
            string getSurname();
private:
            string surname; int phone; int age;
};
```

- Lokalne (utility) funkcije:

Često je korisno deklarisati jednu ili više funkcija kao private, i ove funkcije mogu se koristiti samo unutar same klase. To su tzv. ‘utility functions’ (“komunalne” ili “lokalne” funkcije).

U nastavku je primer korišćenja lokalnih funkcija:

```
class Friend
{
public:
            string getSurname()
            {
                        return surname;
            };
            int getPhone()
            {
                        return phone;
            };
            Friend(string sur0, int ph0)
            {
                        surname = sur0;
                        phone = ph0;
            };
            ~Friend(){};
private:
            void setString(string sur)
            {
                        surname = sur;
            };
            void setPhone(int ph)
            {
                        phone = ph;
            };
            string surname;
            int phone;
};
```

Prijateljske funkcije i klase

prijateljske funkcije, prijateljske klase, friend, sakrivanje podataka

- *Uvod u prijateljske funkcije i klase*
- *Upotreba prijateljskih funkcija*
- *Prijateljske klase*

02

UVOD U PRIJATELJSKE FUNKCIJE I KLASE

Prijateljske funkcije u C++-u su specijalna vrsta funkcija koje imaju pristup privatnim i zaštićenim članovima klase.

Prijateljske funkcije klase su definisane van opsega klase ali imaju prava pristupa svim privatnim (private) i zaštićenim (protected) članovima klase. Iako se prototipovi prijateljskih funkcija navode u okviru definicije klase, treba znati da prijateljske funkcije nisu članice klase.

Prijatelji klase mogu biti: funkcije, šabloni funkcije, funkcije članice druge klase, a takođe i klase i šabloni klase (u slučaju da je cela klasa zajedno sa svim njenim funkcijama članicama prograšena za prijateljsku).

Prijateljske funkcije

Kao što smo spomenuli, prijateljske funkcije u C++-u su specijalna vrsta funkcija koje imaju pristup privatnim i zaštićenim članovima klase. One se smatraju funkcijama koje narušavaju objektno orijentisani princip u C++-u ali postoje slučajevi kada njihovo korišćenje može biti od koristi, npr. kod preklarapanja operatora.

Da bi se neka funkcija proglasila za prijateljskom, neophodno je u okviru definicije klase navesti deklaraciju (prototip) prijateljske funkcije kojoj prethodi ključna reč friend, kao što je prikazano u nastavku:

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

Prijateljske funkcije imaju sledeće osobine:

1) Prijatelji klase mogu biti funkcije članice neke druge klase.
2) Prijatelj jedne klase može biti prijatelj neke druge klase ili svih ostalih klasa u programu, i tada je takav prijatelj poznat kao globalni prijatelj.
3) Prijatelj može da pristupi privatnim ili zaštićenim članovima one klasa za koju je definisano prijateljstvo, ali mogu koristiti članice za specifične objekte.
4) Prijatelji nisu članovi klase pa stoga nemaju pokazivač “this”.
5) Prijatelji mogu biti prijatelji sa više klasa, pa stoga mogu biti korišćeni za razmenu poruka među tim klasama.
6) Prijatelji mogu biti definisani bilo gde (u public, protected ili private sekciji) u klasi.

UPOTREBA PRIJATELJSKIH FUNKCIJA

Da bi se neka funkcija proglasila prijateljskom neophodno je u okviru definicije klase navesti njenu deklaraciju, kojoj prethodi ključna reč friend

U sledećem primeru imamo prijateljsku funciju print koja je članica klase TWO i ima pristup privatnim članovima a i b klase ONE.

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

//Must be known to TWO
//before declaration of ONE.
class ONE;

class TWO
{
public:
              void print(ONE& x);
};

class ONE
{
              int a, b;
              friend void TWO::print(ONE& x);
public:
              ONE() : a(1), b(2) { }
};

void TWO::print(ONE& x)
{
              cout << "a is " << x.a << endl;
              cout << "b is " << x.b << endl;
}
int main()
{
 ONE xobj;
 TWO yobj;
 yobj.print(xobj);

 return 0
}
```

U nastavku je dat još jedan primer korišćenja prijateljskih funkcija

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

class Box
{
   double width;
public:
   friend void printWidth( Box box );
   void setWidth( double wid );
};

// Member function definition
void Box::setWidth( double wid )
{
    width = wid;
}

// Note: printWidth() is not a member function of any
class.
void printWidth( Box box )
{
   /* Because printWidth() is a friend of Box, it can
    directly access any member of this class */
   cout << "Width of box : " << box.width <<endl;
}

// Main function for the program
int main( )
{
   Box box;
   // set box width without member function
   box.setWidth(10.0);

   // Use friend function to print the wdith.
   printWidth( box );

   return 0;
}
```

PRIJATELJSKE KLASE

Kada kreiramo prijateljsku klasu onda sve njene funkcije članice postaju prijateljske funkcije te druge klase.

Osim prijateljskih funkcija, jedna klasa takođe može biti prijateljska nekoj drugoj klasi. Kada kreiramo prijateljsku klasu onda sve njene funkcije članice postaju prijateljske funkcije te druge klase. Ovo zahteva da je ispunjen uslov da klasa koja postaje prijatelj mora prethodno biti deklarisana ili definisana (forward declaration).

Da bi ste sve funkcije članice klase ClassTwo definisali kao prijateljske funkcije klase ClassOne, treba samo navesti sledeću deklaraciju u okviru klase ClassOne:

```
friend class ClassTwo;
```

Pogledajmo sledeći primer:

```
class MyClass
{
            // Declare a friend class
            friend class SecondClass;
public:
            MyClass() : Secret(0){}
            void printMember()
            {
                        cout << Secret << endl;
            }
private:
            int Secret;
};
```

U prethodnoj definiciji vidimo da je klasa SecondClass postavljena kao prijateljska klasi MyClass. Definicija klase SecondClass i glavne funkcije main() je data u nastavku:

```
class SecondClass
{
public:
            void change( MyClass& yourclass, int x )
            {
                        yourclass.Secret = x;
            }
};

void main()
{
            MyClass my_class;
            SecondClass sec_class;
            my_class.printMember();
            sec_class.change( my_class, 5 );
            my_class.printMember();
}
```

Statički članovi klase

statički podaci, statičke funkcije, static

- *Statički podaci klase*
- *Rad sa statičkim podacima objekata*
- *Statičke funkcije članice klase*
- *Upotreba statičkih podataka i funkcija klase*

03

STATIČKI PODACI KLASE

Kada se član klase deklariše kao static to znači da bez obzira koliko se bude kreiralo objekata te klase, kreiraće se samo jedna kopija statičkog člana

Ponekad, neki isti podatak je potreban za sve članove klase. Nepotrebno je, i nefikasno bi bilo da se ovakva vrednost (podatak) memoriše u svakom objektu ponaosob. Zato, takav podatak se može proglasiti „statičkim“ tako što se stavi ključna reč *static* na početku deklaracije polja klase. Kada se podatak klase deklariše kao static to znači da bez obzira koliko se bude kreiralo objekata te klase, kreiraće se samo jedna kopija statičkog člana.

Statičko polje dele svi objekti klase. Svi statički podaci klase se inicijalizuju sa nulom pri kreiranju prvog objekta te klase, ukoliko ne postoji konstruktor koji inicijalizuje drugu vrednost. Inicijalizacija statičkog člana se ne može izvršiti u okviru definicije klase, ali možemo to uraditi izvan klase, kao što je to urađeno u sledećem primeru – korišćenjem operatora pristupa (scope resolution) **::** uz ime klase kako bi znali kojoj klasi odgovarajući podatak pripada. Sintaksa je sledeća:

```
class Xclass
{
public:
          static int n; //deklarise se n kao staticki clan
};
int Xclass::n =0; //globalna definicija n
```

U prethodnom primeru smo inicijalizovali promenljivu n da je jednaka 0, ali pošto se statičke promenljive automatski inicijalizuju sa 0, onda je eksplicitna inicijalizacija potrebna samo ako hoćemo da zadamo vrednost različitu od 0.

RAD SA STATIČKIM PODACIMA OBJEKATA

*Inicijalizacija statičkog člana se vrši izvan definicije klase korišćenjem operatora pristupa **::** uz ime klase kako bi znali kojoj klasi odgovarajući podatak pripada*

Pogledajmo sada sledeći primer da bi smo bolje razumeli funkcionisanje statičkih podataka klase. Definicija klase može biti napisana na sledeći način:

```
#include <iostream>
using namespace std;
class Box
{
   public:
      static int objectCount;
      // 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;
         // Increase every time object is created
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
```

Inicijalizacija statičke promenljive i glavna funkcija main mogu biti napisani na sledeći način:

```
// Initialize static member of class Box
int Box::objectCount = 0;

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

   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount <<
endl;

   return 0;
}
```

Rezultat programa biće:

```
Constructor called.
Constructor called.
Total objects: 2
```

STATIČKE FUNKCIJE ČLANICE KLASE

Statičke funkcije klase su one funkcije koje su nezavisne od objekata klase, i koje mogu biti pozvane iako nije deklarisan nijedan objekat te klase

Deklarisanjem funkcije članice kao statičke, korišćenjem ključne reči static, ona postaje funkcija koja je nezavisna od objekata klase. Neka je deklarisana funkcija Box na sledeći način:

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

class Box
{
   public:
      static int objectCount;
      // 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;
         // Increase every time object is created
         objectCount++;
      }
      double Volume()
      {
         return length * breadth * height;
      }
      static int getCount()
      {
         return objectCount;
      }
   private:
      double length;     // Length of a box
      double breadth;    // Breadth of a box
      double height;     // Height of a box
};
```

Statičke funkcije članice mogu biti pozvane iako nije deklarisan nijedan objekat te klase. Pozivanje statičkih funkcija klase je moguće uraditi korišćenjem naziva klase za kojim sledi operator pristupa (scope resolution operator) **::** i naziv statičke funkcije.

Statičke funkcije mogu da pristupe samo statičkim podacima klase, drugim statičkim funkcijama klase, i svim funkcijama definisanim van klase. Inicijalizacija statičke promenljive van deklaracije klase i pokretačka main funkcija mogu biti napisani na sledeći način:

```
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void)
{
     // Print total number of objects before creating
object.
   cout << "Inital Stage Count: " << Box::getCount() <<
endl;

   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2

   // Print total number of objects after creating object.
   cout << "Final Stage Count: " << Box::getCount() <<
endl;

   return 0;
}
```

UPOTREBA STATIČKIH PODATAKA I FUNKCIJA KLASE

Statičke funkcije mogu da pristupe samo statičkim podacima klase, drugim statičkim funkcijama klase, i svim funkcijama definisanim van klase

Pogledajmo klasu Xclass koja ima automatski konstruktor i destruktor koji ažuriraju broj objekata koji postoje u klasi Xclass, i ima statičku promenljivu n:

```
class Xclass
{
public:
          Xclass(){++n;}
          ~Xclass(){--n;}
          static int n;
};
int Xclass::n=0;
void main ()
{
          Xclass w,y;
          cout << w.n << endl;
          cout << Xclass::n << endl;
}
```

Izlaz bi bio:

```
2
2
```

Takođe se može definisati statička funkcija klase koja je nezavisna od objekata (tj. akcija ove funkcije je nezavisna od objekata) npr.:

```
class Xclass
{
public:
          Xclass(){++n;}
          ~Xclass(){--n;}
          static int nvalueF() {return n;}

private:
          static int n;
};

int Xclass::n = 0;

void main ()
{
          cout << Xclass::nvalueF() << endl;
          Xclass w,y;
          cout << Xclass::nvalueF() << endl;
}
```

Nakon izvršenja programa dobija se sledeći izlaz:

```
0
2
```

Anonimne promenljive i objekti

anonimne promenljive, anonimni objekti

- *Uvodna razmatranja*
- *Anonimni objekti klase*
- *Upotreba anonimnih objekata*

04

UVODNA RAZMATRANJA

Anonimna promenljiva je ona promenljiva kojoj nije dodeljeno ime. Anonimne promenljive imaju takozvani "expression scope", što znači da se uništavaju na kraju izraza u kome se kreiraju

Postoje slučajevi u kojima nam je neka promenljiva potrebna samo privremeno. Uzmimo u obzir, na primer, sledeću situaciju:

```
int Add(int nX, int nY)
{
    int nSum = nX + nY;
    return nSum;
}

int main()
{
    using namespace std;
    cout << Add(5, 3);

    return 0;
}
```

Primetimo da u funkciji Add() promenljiva nSum se koristi samo privremeno da se izračuna vrednost zbira dva broja. Ona nema neku veliku ulogu, i njena funkcija je samo da se vrednost sumiranja prosledi do linije u kojoj se izračunata vrednost vraća pozivaocu (do naredbe return).

Postoji jedan lakši način da se napiše funkcija Add() korišćenjem anonimne promenljve. **Anonimna promenljiva** je ona promenljiva kojoj nije dodeljeno ime. Anonimne promenljive u C++-u imaju takozvani "expression scope", što znači da se uništavaju na kraju izraza u kome se kreiraju. Stoga, njihovo korišćenje mora biti ostvareno istog trenutka! U nastavku je prikazana funkcija Add() napisana korišćenjem anonimne promenljive:

```
int Add(int nX, int nY)
{
    return nX + nY;
}
```

Kada se izračuna izraz nX + nY, rezultat se smešta u anonimnu (neimenovanu) promenljivu. Kopija anonimne promenljive se vraća pozivaocu funkcije po vrednosti. Ovo se ne radi samo sa povratnim vrednostima funkcije već i sa parametrima funkcije.

Pretpostavimo da imamo sledeći primer:

```
void PrintValue(int nValue)
{
    using namespace std;
    cout << nValue;
}
int main()
{
    int nSum = 5 + 3;
    PrintValue(nSum);
    return 0;
}
```

Umesto prethodnog možemo napisati poziv funkcije PrintValue na sledeći način:

```
int main()
{
    PrintValue(5 + 3);
    return 0;
}
```

Primetimo kako naš kod sada izgleda mnogo čistije jer nismo bespotrebno deklarisali pomoćne promenljive koje ćemo da koristimo samo jednom.

ANONIMNI OBJEKTI KLASE

Deklarisanje anonimnog objekta se ostvaruje uobičajenim kreiranjem objekta s tim što se izostavlja ime promenljive

Kao u prethodnom primeru, gde smo radili sa primitivnim tipovima podataka C++ jezika, moguće je na sličan način deklarisati anonimni objekat korisnički definisane klase. Ovo se ostvaruje uobičajenim kreiranjem objekta s tim što se izostavlja ime promenljive, kao u sledećem primeru:

```
Cents cCents(5); // normal variable
Cents(7); // anonymous variable
```

U prethodnom isečku koda iskaz Cents(7) će kreirati anonimni objekat klase Cents, inicijalizovaće nekog njegovog člana vrednošću 7, a zatim će ga uništiti. U ovakvom primeru, možemo primetiti, korišćenje anonimnog objekta nema nikakvog smisla. Stoga, pogledajmo sledeći primer gde anonimni objekti mogu biti od koristi:

```
class Cents
{
private:
    int m_nCents;
public:
    Cents(int nCents) { m_nCents = nCents; }
    int GetCents() { return m_nCents; }
};
Cents Add(Cents &c1, Cents &c2)
{
    Cents cTemp(c1.GetCents() + c2.GetCents());
    return cTemp;
}
```

Glavni program ćemo napisati na sledeći način:

```
int main()
{
    Cents cCents1(6);
    Cents cCents2(8);
    Cents cCentsSum = Add(cCents1, cCents2);
    std::cout << "I have " << cCentsSum.GetCents()
            << " cents." << std::endl;
    return 0;
}
```

Primetimo da je ovaj primer dosta sličan primeru koji smo radili sa celim brojevima u sekciji Uvodna razmatranja. U ovom slučaju imamo funkciju Add() u kojoj se definiše promenljiva kratkog veka cTemp koja služi samo za kreiranje novog objekta koji će istog trenutka biti vraćen naredbom return pozivaocu funkcije Add().Takođe se može primetiti da koristimo i pomoćnu promenljivu cCentsSum u glavnoj funkciji main(), koja ima ulogu da prihvati vrednost funkcije koju zatim štampamo na ekranu.

UPOTREBA ANONIMNIH OBJEKATA

Anonimni objekti se primarno upotrebljavaju da bi se prosledila ili prihvatila vrednost pri radu sa funkcijama, bez kreiranja puno pomoćnih promenljivih koje bi služile u istu svrhu

Prethodni program se može znatno uprostiti korišćenjem anonimnih promenljivih, što je prikazano u nastavku:

```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    int GetCents() { return m_nCents; }
};

Cents Add(Cents &c1, Cents &c2)
{
    return Cents(c1.GetCents() + c2.GetCents());
}

int main()
{
    Cents cCents1(6);
    Cents cCents2(8);
    std::cout << "I have " << Add(cCents1,
cCents2).GetCents()
<< " cents." << std::endl;

    return 0;
}
```

Ova verzija funkcije Add() radi identične operacije kao i prethodna, s tim što koristimo anonimnu promenljivu tipa Cents umesto imenovane promenljive. Takođe se može primetiti u funkciji main() da ne koristimo promenlivu cCentsSum koja služi za prihvatanje rezultata funkcije. Umesto toga, koristimo anonimnu povratnu vrednost funkcije Add(), čije podatke odmah štampamo na ekran!

Kao rezultat, naš program je kraći, čistiji, i lakši za praćenje (od trenutka kada usvojimo znanje o konceptu anonimnih promenljivih).

U C++-u, anonimne promenljive se primarno upotrebljavaju da bi se prosledila ili prihvatila vrednost pri radu sa funkcijama, bez potrebe kreiranja puno pomoćnih promenljivih koje bi služile u istu svrhu. Međutim, ponekad nam ništa ne znači da anonimni objekat prosledimo ili vratimo po vrednosti! Ukoliko je promenljiva prosleđena ili vraćena po referenci ili adresi onda je neophodno koristiti imenovane promenljive. Takođe, s obzirom da anonimne promenljive imaju opseg važenja u izrazu u kome se koriste, ukoliko je neophodno referencirati vrednost koja se javlja u višestrukim izrazima opet je neophodno koristiti imenovane promenljive.

Uvod u preklapanje funkcija i operatora

preklapanje funkcija, preklapanje operatora, overloadiing

- *Preklapanje funkcija i operatora*
- *Preklapanje funkcija*
- *Preklapanje operatora (operator overloading)*
- *Primeri preklapanja operatora*

05

PREKLAPANJE FUNKCIJA I OPERATORA

Predefinisana (preklopljena) deklaracija je ona deklaracija koja ima isti naziv kao i neka prethodna deklaracija s tim što deklaracije imaju različite argumente i naravno različitu definiciju

C++ dozvoljava definisanje više od jedne funkcije istog naziva kao i više operatora, što se naziva preklapanje funkcija odnosno preklapanje operatora, respektivno. Osim termina preklapanje često se koristi i termin predefinisanje (overloading).

Predefinisana (preklopljena) deklaracija je ona deklaracija koja ima isti naziv kao i neka prethodna deklaracija (pričamo ovde o istom opsegu važenja), s tim što deklaracije imaju različite argumente i naravno različitu definiciju (implementaciju).

Kada se vrši pozivanje preklopljene funkcije ili operatora, kompajler određuje koja je pogodnija definicija koju treba da koristi, na osnovu poređenja tipova argumenata ili poređenja operatora sa tipom parametara koji su navedeni u definiciji. Proces izbora najpogodnije preklopljene funkcije ili operatora se naziva **razrešavanje preklapanja**. Operatori koji se mogu preklopiti su:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Slika-1 Operatori programskog jezika C++ koji se mogu preklopiti (predefinisati)

Međutim, postoje operatori koji se ne mogu preklopiti i oni su prikazani na sledećoj slici:

::	.*	.	?:

Slika-2 Spisak operatora koji se ne mogu predefinisati (preklopiti)

PREKLAPANJE FUNKCIJA

C++ dozvoljava postojanje više funkcija sa istim imenom, s tim što se definicije funkcija moraju međusobno razlikovati po tipu i broju argumenata

C++ dozvoljava postojanje više funkcija sa istim imenom, u okviru istog opsega (tj. u okviru definicije iste klase, istog imenskog prostora, itd…). Definicije funkcija se moraju međusobno razlikovati po tipu i broju argumenata u listi argumenata funkcije. Nije moguće izvršiti preklapanje funkcija ako se funkcije razlikuju samo po tipu rezultata, odnosno tipu povratne vrednosti (return type). U nastavku je primer gde imamo funkcije sa istim nazivom **print()** koje se koriste za štampanje različitih tipova podataka:

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

class printData
{
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }

      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }

      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};
```

Glavni program može biti napisan na sledeći način

```
int main(void)
{
   printData pd;

   // Call print to print integer
   pd.print(5);
   // Call print to print float
   pd.print(500.263);
   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

Rezultat prethodnog programa će biti:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

PREKLAPANJE OPERATORA (OPERATOR OVERLOADING)

Preklopljeni operatori su ustvari funkcije specijalnog naziva koji se sastoji iz ključne reči operator za kojim sledi naziv operatora koji želimo da preklopimo

U C++-u je moguće predefinisati, tj. preklopiti najveći deo standardnih ugrađenih (build-in) tipova operatora. Na taj način korisnik može da koristi preklopljene operatore nad novim tipovima koje definiše. Preklopljeni operatori su ustvari funkcije specijalnog naziva koji se sastoji iz ključne reči operator za kojim sledi naziv operatora koji želimo da preklopimo. Kao i bilo koja druga funkcija, preklopljeni operator ima povratnu vrednost i listu parametara.

Kada se govori o preklapanju operatora misli se na standardne operatore koje smo upoznali u okviru sintakse C jezika. U C++ jeziku moguće je predefinisati (preklopiti) funkciju standardnih operatora tako da novoformirani operator kao argumente koristi objekte i to na način kako to programer želi. Odmah uviđamo da operatori moraju biti ili funkcije članice klasa ili imati klasu kao barem jedan od parametara.

Kada operator nije član klase, sintaksa za preklapanje operatora je:

```
tip operator naziv_operatora ( parametri )
{
            // sadrzaj - programski kod funkcije
}
```

Kada je operator član klase, sintaksa je:

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

Da bi se shvatio mehanizam kojim se vrši preklapanje operatora, razmotrićemo kako kompajler tretira jednostavan izraz. Izraz tipa a+b kompajler će interno predstaviti na sledeći način:

```
a.operator +(b)
```

Operator + je u suštini funkcija članica klase kojoj objekat a pripada, dok je b argument te funkcije. U nastavku je prikazana deklaracija predefinisanog operatora sabiranja + klase Box:

```
Box operator+(const Box&);
```

koji se može korisiti da bi se sabrala dva objekta tipa Box i kao rezultat funkcije će da se dobije novi objekat tipa Box. Najveći deo preklopljenih operatora se mogu definisati kao obične funkcije koje jesu ili nisu članice klase. U slučaju da prethodnu funkciju deklarišemo kao ne-članicu klase onda je neophodno da joj prosledimo dva parametra za svaki od objekata, kao što sledi:

```
Box operator+(const Box&, const Box&);
```

PRIMERI PREKLAPANJA OPERATORA

U C++-u je moguće predefinisati, tj. preklopiti najveći deo standardnih ugrađenih tipova operatora

U nastavku će biti opisani sledeći primeri preklapanja operatora koji će pomoći da se bolje razume ceo koncept:

- Preklapanje unarnih operatora
- Preklapanje binarnih operatora
- Preklapanje operatora poređenja
- Preklapanje ulazno/izlaznih operatora
- Preklapanje operatora uvećanja i umanjenja (++ i --)
- Preklapanje operatora dodele vrednosti
- Preklapanje operatora poziva funkcije ()
- Preklapanje operatora []

Treba imati na umu da nije moguće kreirati nove operatore; moguće je samo preklopiti već postojeće. Nažalost, ovo sprečava programera da koristi popularne oblike operatora kao što je ** koji služi za stepenovanje.

Takođe treba imati na umu da preklapanje operatora dodele i operatora sabiranja kako bi se mogao koristiti sledeći iskaz nad objektima:

```
object2 = object2 + object1;
```

ne znači da je automatski dozvoljeno i korišćenje kombinovanog operatora dodele i sabiranja +=, tj.

```
object2 += object1;
```

Operatori osnovnih tipova

Značenje, odnosno, način kako neki operator deluje na objekte osnovnih tipova ne može biti promenjeno ili predefinisano. To znači da programer ne može, na primer, da izmeni način na koji operator + sabira dva cela broja. Predefinisanje operatora je moguće uraditi samo za korisnički definisane tipove podataka, ili za mešavinu objekta korisnički definisanog i osnovnog tipa.

Preklapanje unarnih operatora

Unarni operatori se izvršavaju nad jednim operandom i u nastavku su dati primeri unarnih operatora:

- Operator inkrementiranja (++) i dekrementiranja (--).
- Unarni operator minus (-).
- Operator logičke negacije (!).

Unarni operatori se izvršavaju nad objektom za koji su pozvani i, normalno, ovi operatori se pojavljuju ili sa leve strane objekta, kao na primer: !obj, -obj, i ++obj odnosno ponekad i sa desne strane objekta kao na primer u slučaju: obj++ ili obj--.

Preklapanje unarnih i binarnih operatora

predefinisanje, unarni operatori, binarni operatori,

- *Preklapanje operatora umanjenja ++ i uvećanja --*
- *Preklapanje unarnog operatora (-)*
- *Preklapanje binarnih operatora*
- *Primer preklapanja operatora sabiranja +*
- *Preklapanje operatora korišćenjem prijateljskih funkcija*
- *Preklapanje operatora korišćenjem prijateljskih funkcija*

06

PREKLAPANJE OPERATORA UMANJENJA ++ I UVEĆANJA --

Razlika između deklaracije preklapanja prefiksnog i postfiksnog operatora inkrementiranja je u navođenju argumenata funkcije preklapanja operatora

U nastavku je dat primer kako je moguće preklopiti operator inkrementiranja u prefiksnoj (++a) ili u postfiksnoj (a++) notaciji. Na sličan način je moguće preklopiti i operator umanjenja (--).

```
class Time
{
   private:
      int hours;             // 0 to 23
      int minutes;           // 0 to 59
   public:

      Time(){  // required constructors
         hours = 0;
         minutes = 0;
      }
      Time(int h, int m){
         hours = h;
         minutes = m;
      }
      void displayTime() // method to display time
      {
         cout << "H: " << hours << " M:" << minutes <<endl;
      }
      Time operator++ ()  // overloaded prefix ++ operator
      {
         ++minutes;          // increment this object
         if(minutes >= 60)
         {
            ++hours;
            minutes -= 60;
          }
      return T;  // return old original value
      }
};
```

Glavni program ćemo napisati na sledeći način:

```
int main()
{
   Time T1(11, 59), T2(10,40);

   ++T1;                     // increment T1
   T1.displayTime();         // display T1
   ++T1;                     // increment T1 again
   T1.displayTime();         // display T1

   T2++;                     // increment T2
   T2.displayTime();         // display T2
   T2++;                     // increment T2 again
   T2.displayTime();         // display T2
   return 0;
}
```

Rezultat će biti:

```
H: 12 M:0
H: 12 M:1
H: 10 M:41
H: 10 M:42
```

U prethodnoj klasi možemo da primetimo razliku između načina definisanja prefiksnog i postfiksnog operatora (++a i a++)

```
Time operator++ ();   // prefix ++
Time operator++(int); // postfix ++
```

PREKLAPANJE UNARNOG OPERATORA (-)

Deklaracija preklapanja unarnog operatora (-) ima oblik: „ImeKlase operator -();“

U nastavku je dat primer kako je moguće izvršiti preklapanje operatora minus (-) u prefiksnoj (-a) notaciji.

```
class Distance
{
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      // method to display distance
      void displayDistance()
      {
         cout << "F: " << feet << " I:" << inches
<<endl;
      }
      // overloaded minus (-) operator
      Distance operator- ()
      {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches);
      }
};
```

Glavni program ćemo napisati na sledeći način:

```
int main()
{
   Distance D1(11, 10), D2(-5, 11);

   -D1;                      // apply negation
   D1.displayDistance();     // display D1

   -D2;                      // apply negation
   D2.displayDistance();     // display D2

   return 0;
}
```

Nakon kompajliranja i izvršavanja programa dobija se sledeći rezultat

```
F: -11 I:-10
F: 5 I:-11
```

Ukoliko ste uspeli da uhvatite poentu prethodnog primera, moguće je sličan princip primeniti na preklapanje logičkog Ne operatora (! - NOT).

PREKLAPANJE BINARNIH OPERATORA

Binarni operatori su oni koji se obavljaju nad dva argumenta. Najčešće korišćeni binarni operatori su operatori sabiranja (+), oduzimanja (-), množenja (*) i deljenja (/).

Binarni operatori su oni koji se obavljaju nad dva argumenta. Najčešće korišćeni binarni operatori su operatori sabiranja (+), oduzimanja (-), množenja (*) i deljenja (/). U nastavku je dat primer preklapanja operatora sabiranja (+). Na isti način je moguće izvršiti preklapanje operatora oduzimanja (-) i deljenja (/)

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

   double getVolume(void) {return length * breadth *
height;}
   void   setLength( double len ) {length = len;}
   void   setBreadth( double bre ){breadth = bre;}
   void   setHeight( double hei ) {height = hei;}

   // Overload + operator to add two Box objects.
   Box operator+(const Box& b)
   {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
   }
};
```

```
// Main function for the program
int main( )
{
   Box Box1;     // Declare Box1 of type Box
   Box Box2;     // Declare Box2 of type Box
   Box Box3;     // Declare Box3 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();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

PRIMER PREKLAPANJA OPERATORA SABIRANJA +

Osnovni oblik preklapanja operatora sabiranja je „Tip operator + (const Tip& a)“

U narednom primeru imamo klasu Box, i predefinisani operator + gde je jedan od objekata prosleđen kao argument čijim svojstvima (poljima) se pristupa preko ovog objekta dok se drugom objektu pristupa preko pokazivača this:

```
class Box
{
   public:
      double getVolume(void)
      {
         return length * breadth * height;
      }
      void setLength( double len )
      {
          length = len;
      }
      void setBreadth( double bre )
      {
          breadth = bre;
      }
      void setHeight( double hei )
      {
          height = hei;
      }
      // Overload + operator to add two Box objects.
      Box operator+(const Box& b)
      {
         Box box;
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
         return box;
      }
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

Glavna main funkcija ima sledeći oblik:

```
int main( )
{
   Box Box1;        // Declare Box1 of type Box
   Box Box2;        // Declare Box2 of type Box
   Box Box3;        // Declare Box3 of type Box
   double volume = 0.0;
   // 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();
   cout << "Volume of Box2 : " << volume <<endl;
   // Add two object as follows:
   Box3 = Box1 + Box2;
   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;
   return 0;
}
```

Rezultat prethodnog programa će biti:
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400

PREKLAPANJE OPERATORA KORIŠĆENJEM PRIJATELJSKIH FUNKCIJA

Funkcije-operacije, osim što se mogu realizovati kao članovi klase, mogu biti i prijateljske funkcije klase. Prijateljske funkcije ne mogu preklopiti operacije =, (), [], ->

Kada operator ne modifiikuje operande nad kojima se primenjuje onda je najbolji način da se preklapanje izvrši korišćenjem prijateljske funkcije. Nijedan od aritmetičkih operatora ne modifukuje svoje operande (oni samo proizvode i vraćaju rezultat), tako da ćemo koristiti prijateljske funkcije za preklapanje aritmetičkih operatora.

Sledeći primer pokazuje kako se vrši preklapanje operatora sabiranja (+) u cilju sabiranja dva objekta tipa "Cents":

```
class Cents
{
private:
            int m_nCents;
public:
            Cents(int nCents) { m_nCents = nCents; }
            // overload Cents + Cents
            friend Cents operator+(const Cents &c1,
const Cents &c2);
            // overload Cents - Cents
            friend Cents operator-(const Cents &c1,
const Cents &c2);
            int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
            // use the Cents constructor and
operator+(int, int)
            return Cents(c1.m_nCents + c2.m_nCents);
}
```

```
// note: this function is not a member
function!
Cents operator-(const Cents &c1, const Cents
&c2)
{
 // use the Cents constructor and operator-
(int, int)
 return Cents(c1.m_nCents - c2.m_nCents);
}
int main()
{
 Cents cCents1(6);
 Cents cCents2(8);
 Cents cCentsSum = cCents1 + cCents2;
 std::cout << "I have " << cCentsSum
.GetCents()
<< " cents." << std::endl;
 return 0;
}
```

PREKLAPANJE OPERATORA KORIŠĆENJEM PRIJATELJSKIH FUNKCIJA

Funkcije-operacije, osim što se mogu realizovati kao članovi klase, mogu biti i prijateljske funkcije klase. Prijateljske funkcije ne mogu preklopiti operacije =, (), [], ->

Prethodni primer će proizvesti sledeći rezultat:

I have 14 cents.

Preklapanje operatora sabiranja (+) je prosto kao i deklarisanje funkcije koja ima naziv operator+, prosleđujući joj dva parametra tipa operanada koje hoćemo da saberemo, za koju biramo odgovarajući povratni tip, i zatim pišemo definiciju funkcije.

U slučaju našeg objekta tipa Cents, implementacija funkcije operator+() se sastoji iz tri koraka. Prvo, biramo tip parametara: u ovoj verziji funkcije operator+, sabiraćemo dva objekta tipa Cents, tako da će funkcija prihvatiti dva objekta tipa Cents. Drugo, treba da izaberemo povratni tip: funkcija operator+ će kao rezultat vratiti neki objekat tipa Cents, pa će to biti povratni tip. Konačno, treba implementirati funkciju: da bi smo sabrali dva objekta tipa Cents, neophodno je sabrati članove m_nCents svih Cents objekata. Pošto je preklopljena funkcija operator+() prijateljska sa klasom Cents, moguće je direktno pristupiti članovima klase m_nCents. Takođe, pošto je promenljiva m_nCents tipa int, i pošto C++ zna kako da sabere dva cela broja, ostaje nam da samo iskoristimo operator + kako bi izvršili sabiranje.

Na sličan način se mogu preklopiti operacije deljenja i množenja.

Preklapanje operatora poređenja i dodele

Preklapanje, operatori poredjenja, operator dodele

- *Preklapanje operatora poređenja*
- *Preklapanje operatora dodele vrednosti =*

07

PREKLAPANJE OPERATORA POREĐENJA

Korisnici imaju mogućnost da preklope bilo koji od operatora poređenja (<, >, <=, >=, ==, itd.) u cilju poređenja objekata neke klase

U programskom jeziku C++ moguće je koristiti različite operatore poređenja kao što su (<, >, <=, >=, ==, itd.) u cilju poređenja ugrađenih (primitivnih) C++ tipova podataka.

Korisnici imaju mogućnost da preklope bilo koji od ovih operatora u cilju poređenja objekata neke klase.

U nastavku je dat primer kako je moguće preklopiti operator poređenja <, a korišćenjem ovog principa moguće je preklopiti i sve ostale operatore poređenja.

```
class Distance
{
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      // method to display distance
      void displayDistance()
      {
         cout << "F: " << feet << " I:" << inches
<<endl;
      }
}
```

Glavni program možemo napisati kao:

```
int main()
{
   Distance D1(11, 10), D2(5, 11);

   if( D1 < D2 )
   {
      cout << "D1 is less than D2 " << endl;
   }
   else
   {
      cout << "D2 is less than D1 " << endl;
   }
   return 0;
}
```

Rezultat će biti:

```
D2 is less than D1
```

PREKLAPANJE OPERATORA DODELE VREDNOSTI =

Kad god u klasi treba definisati konstruktor kopiranja, njega treba da prati operatorska funkcija dodeljivanja, i obrnuto

Posmatrajmo sledeću definiciju klase Distance:

```
class Distance
{
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      void operator=(const Distance &D )
      {
         feet = D.feet;
         inches = D.inches;
      }
      // method to display distance
      void displayDistance()
      {
         cout << "F: " << feet <<  " I:" <<  inches
<< endl;
      }
};
```

Preklopljeni operator dodele može biti korišćen pri kreiranju objekata na sličan način kako to radi konstruktor kopiranja. Glavni program može biti napisan kao:

```
int main()
{
   Distance D1(11, 10), D2(5, 11);

   cout << "First Distance : ";
   D1.displayDistance();
   cout << "Second Distance :";
   D2.displayDistance();

   // use assignment operator
   D1 = D2;
   cout << "First Distance :";
   D1.displayDistance();

   return 0;
}
```

Rezultat će biti

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

U C++-u je moguće preklopiti operator dodele (=) na isti način kako se vrši preklapanje ostalih operatora.

Preklapanje ulazno/izlaznih operatora

Preklapanje operatora, ubacivanje u tok, izvlačenje iz toka

- *Osnovi o preklapanju ulazno/izlaznih operatora*
- *Primer preklapanja ulazno/izlaznih operatora*

08

OSNOVI O PREKLAPANJU ULAZNO/IZLAZNIH OPERATORA

Operatori ubacivanja u tok i izvlačenja iz toka mogu biti preklopljeni kako bi se obezbedio ulaz/izlaz korisnički definisanih tipova kao što su objekti.

Za klase koje imaju veliki broj podataka članova, štampanje na ekran svake od članice posebno će brzo postati zamorno. Uzmimo u obzir sledeću klasu:

```
class Point
{
private:
    double m_dX, m_dY, m_dZ;

public:
    Point(double dX=0.0, double dY=0.0, double
dZ=0.0)
    {
    m_dX = dX;
    m_dY = dY;
    m_dZ = dZ;
    }

    double GetX() { return m_dX; }
    double GetY() { return m_dY; }
    double GetZ() { return m_dZ; }
};
```

Ako želimo da oštampamo instance klase na ekran, onda možemo napisati nešto slično sledećem:

```
Point cPoint(5.0, 6.0, 7.0);
cout << "(" << cPoint.GetX() << ", " <<
    cPoint.GetY() << ", " <<
    cPoint.GetZ() << ")";
```

i to samo za jednu instancu! Bilo bi mnogo lakše kada bi mogli jednostavno da napišemo:

```
Point cPoint(5.0, 6.0, 7.0);
cout << cPoint;
```

i da dobijemo isti rezultat. Preklapanjem operatora << ovo postaje moguće. Preklapanje operatora << je slično preklapanju operatora sabiranja + (oba operatora su binarni operatori), samo što se tipovi parametara razlikuju.
Uzimimo u obzir sledeći izraz cout << cPoint. Postavlja se pitanje: ako je operand << šta su onda operandi? Levi operand je cout objekat, a desni operator je objekat klase Point. Cout je ustvari jedan objekat klase ostream. Stoga, predefinisana (preklopljena) funkcija će imati sledeću deklaraciju:

```
friend ostream& operator<< (ostream &out, Point &cPoint);
```

U ovom slučaju je bitno kreirati funkciju za preklapanje operatora koja će biti prijateljska funkciji klasi, jer će biti pozvana bez kreiranja objekta.

Preklapanje operatora ubacivanja u tok **>>** je gotovo analogno preklapanju operatora izvlačenja iz toka <<.

PRIMER PREKLAPANJA ULAZNO/IZLAZNIH OPERATORA

Preklapanje izlaznog operatora se najčešće vrši korišćenjem konstantnog argumenta objekta (const ImeKlase& imeObjekta)

Posmatrajmo sledeću definiciju klase Distance:

```
class Distance
{
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      friend ostream &operator<<( ostream &output,
                                        const
Distance &D )
      {
         output << "F : " << D.feet << " I : " <<
D.inches;
         return output;
      }

      friend istream &operator>>( istream  &input,
Distance &D )
      {
         input >> D.feet >> D.inches;
         return input;
      }
};
```

Glavni program za testiranje je:

```
int main()
{
   Distance D1(11, 10), D2(5, 11), D3;
   cout << "Enter the value of object : " <<
endl;
   cin >> D3;
   cout << "First Distance : " << D1 << endl;
   cout << "Second Distance :" << D2 << endl;
   cout << "Third Distance :" << D3 << endl;
   return 0;
}
```

Rezultat programa će biti:

Enter the value of object :
70
10
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
Third Distance :F : 70 I : 10

Preklapanje izlaznog operatora se vrši korišćenjem konstantnog argumenta objekta (const Distance &D). Na ovaj način, moguće je štampati istovremeno konstantne i nekonstanne objekte. Međutim, kod preklapanje ulaznog operatora >>, neophodno je da se podatak cPoint ostavi kao nekonstantan jer, naravno, preklopljeni operator >> menja podatak cPoints.

Preklapanje operatora poziva funkcije ()

Preklapanje operatora, operator poziva funkcije

- *Osnovi o preklapanju operatora poziva funkcije ()*

09

OSNOVI O PREKLAPANJU OPERATORA POZIVA FUNKCIJE ()

Kada se vrši preklapanje operatora ( ) kreira se ustvari funkcijski operator koji može biti prosleđen kao proizvoljan broj parametara

Posmatrajmo sledeću definiciju klase Distance:

```
class Distance
{
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12
   public:
      // required constructors
      Distance(){
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i){
         feet = f;
         inches = i;
      }
      // overload function call
      Distance operator()(int a, int b, int c)
      {
         Distance D;
         // just put random calculation
         D.feet = a + c + 10;
         D.inches = b + c + 100 ;
         return D;
      }
      // method to display distance
      void displayDistance()
      {
         cout << "F: " << feet <<  " I:" <<  inches
<< endl;
      }
}
```

Operator poziva funkcije () može biti preklopljen za objekte klase. Kada se vrši preklapanje operatora ( ), tada se ustvari ne kreira novi način da se pozove funkcija. Naime, kreira se funkcijski operator koji može biti prosleđen kao proizvoljan broj parametara. U nastavku je data pokretačka main funkcija:

```
int main()
{
   Distance D1(11, 10), D2;

   cout << "First Distance : ";
   D1.displayDistance();

   D2 = D1(10, 10, 10); // invoke operator()
   cout << "Second Distance :";
   D2.displayDistance();

   return 0;
}
```

Rezultat prethodnog programa će biti:

```
First Distance : F: 11 I:10
Second Distance :F: 30 I:120
```

Preklapanje operatora indeksiranja []

Preklapanje operatora, operator indeksiranja, nizovi

- *Postupak preklapanja operatora indeksiranja []*
- *Primena preklapanja operatora indeksiranja []*

10

POSTUPAK PREKLAPANJA OPERATORA INDEKSIRANJA []

Operator indeksiranja se obično preklapa kada funkcija sadrži niz elemenata, za koje indeksiranje ima smisla

Kada radimo sa nizovima, obično koristimo operator indeksiranja kako bi pristupili određenom elementu niza

```
anArray[0] = 7; // put 7 in the 1. element
```

Međutim, uzmimo u obzir sledeći primer gde imamo klasu IntList čiji član je niz celih brojeva:

```
class IntList
{
private:
    int m_anList[10];
};

int main()
{
    IntList cMyList;
    return 0;
}
```

Pošto je podatak m_anList privatan član klase, ne možemo mu direktno pristupiti preko objektne promenljive cMyList. Ovo znači da nemamo mogućnost da direktno očitamo ili postavimo vrednost elementima niza m_anList (korišćenjem get i set metoda). Stoga se postavlja pitanje kako da očitamo ili postavimo odgovarajuće vrednosti elementima niza u našoj listi? U slučaju da ne koristimo preklapanje operatora, tipičan način je da kreiramo funkcije pristupa, na sledeći način:

```
class IntList
{
private:
    int m_anList[10];

public:
    void SetItem(int nIndex, int nData) {
m_anList[nIndex] = nData; }
    int GetItem(int nIndex) { return
m_anList[nIndex]; }
};
```

Iako ovo funkcioniše, ipak nije baš korisnički gledano pogodna opcija (user friendly). Posmatrajmo sledeći primer:

```
int main()
{
    IntList cMyList;
    cMyList.SetItem(2, 3);

    return 0;
}
```

Da li vrednost 2. člana niza postavljamo na 3, ili vrednost 3. člana na 2, je nešto što nije do kraja jasno bez gledanja u implementaciju odgovarajuće set metode. Stoga je bolja opcija da se u ovom slučaju izvrši predefinisanje operatora [] kako bi se omogućio pristup elementima niza m_anList. Operator indeksiranja je jedan od operatora koji mora biti preklopljen isključivo korišćenjem funkcija članica. U ovom slučaju, naš predefinisani operator [] imaće jedan argument, ceo broj, koji predstavlja indeks niza a kao razultat će da vrati ceo broj.

PRIMENA PREKLAPANJA OPERATORA INDEKSIRANJA []

Operator indeksiranja je jedan od operatora koji mora biti preklopljen isključivo korišćenjem funkcija članica

Neka je klasa safearray napisana na sledeći način:

```
#include <iostream>
using namespace std;
const int SIZE = 10;

class safearay
{
   private:
      int arr[SIZE];
   public:
      safearay()
      {
         register int i;
         for(i = 0; i < SIZE; i++)
         {
           arr[i] = i;
         }
      }
      int &operator[](int i)
      {
          if( i > SIZE )
          {
              cout << "Index out of bounds" <<endl;
              // return first element.
              return arr[0];
          }
          return arr[i];
      }
};
```

Glavni program za prethodno napisanu klasu možemo napisati kao:

```
int main()
{
   safearay A;

   cout << "Value of A[2] : " << A[2] <<endl;
   cout << "Value of A[5] : " << A[5]<<endl;
   cout << "Value of A[12] : " << A[12]<<endl;

   return 0;
}
```

Nakon izvršavanja prethodnog programa dobiće se sledeći rezultat:

```
Value of A[2] : 2
Value of A[5] : 5
Index out of bounds
Value of A[12] : 0
```

Preklapanje operatora konverzije tipa

Preklapanje operatora, konverzija tipa, kastovanje

- *Uvodna razmatranja*
- *Preklapanje operacije konverzije tipa*
- *Primena preklapanja konverzije tipa*

11

UVODNA RAZMATRANJA

Preklapanje operatora konverzije tipa omogućava konverziju definisane klase u neki drugi tip podatka

U lekciji o konverziji tipova podataka naučili smo da C++ dozvoljava implicitnu i eksplicitnu konverziju jednog tipa u drugi. U nastavku je dat primer koji konvertuje tip int u double:

```
int nValue = 5;
double dValue = nValue; // int implicitly cast to a double
```

C++ već zna kako da konvertuje ugrađene (primitivne) tipove podataka, ali on ne zna kako da konvertuje korisničke tipove podataka. U takvim slučajevima dolazi do izražaja preklapanje operacije konverzije tipa. Preklapanje operatora konverzije tipa omogućava konverziju definisane klase u neki drugi tip podatka. Pogledajmo za početak sledeću klasu Cents:

```
class Cents
{
private:
            int m_nCents;
public:
            Cents(int nCents=0)
            {
                        m_nCents = nCents;
            }

            int GetCents() { return m_nCents; }
            void SetCents(int nCents) { m_nCents
= nCents; }
};
```

Ova klasa je prilično prosta: ona čuva neki broj centi kao celobrojnu (int) vrednost i ima funkcije koje mogu da očitaju ili menjaju podatak o broju centi. Klasa takođe sadrži konstruktor koji inicijalizuje polje klase m_nCents, time što se vrši pretvaranje celih brojeva u cente.

Ukoliko imamo mogućnost da konvertujemo ceo broj u cente, onda ima smisla da obezbedimo mogućnost da konvertujemo cente nazad u neki ceo broj. U narednom primeru, moramo da koristimo funkciju GetCents() da bi smo konvertovali promenljivu klase Cents u ceo broj koji će zatim biti oštampan na ekranu pomoću funkcije PrintInt():

```
void PrintInt(int nValue)
{
            cout << nValue;
}

int main()
{
            Cents cCents(7);
            PrintInt(cCents.GetCents()); // print 7

            return 0;
}
```

Ukoliko već imamo napisan ogroman broj funkcija koje kao parametar prihvataju ceo broj, naš kod će biti zatrpan pozivima funkcije GetCents() što čini veću zbrku nego što treba da bude.

PREKLAPANJE OPERACIJE KONVERZIJE TIPA

Operator kastovanja nema povratni tip. C++ pretpostavlja da će korisnik vratiti korektan tip kao rezultat funkcije

Da bi smo stvari načinili lakšim, izvršićemo preklapanje konverzije tipa int, što će nam omogućiti kastovanje tipa Cents u tip int. U nastavku je primer koji pokazuje kako je to odrađeno:

```
class Cents
{
private:
            int m_nCents;
public:
            Cents(int nCents=0)
            {
                        m_nCents = nCents;
            }

            // Overloaded int cast
            operator int() { return m_nCents; }

            int GetCents() { return m_nCents; }
            void SetCents(int nCents) { m_nCents =
nCents; }
};
```

Postoje dve stvari koje treba naglasiti:

1) Da bi se preklopila funkcija kastovanja naše klase u tip int, napisali smo novu funkciju u okviru naše klase koja se naziva operator int(). Može se primetiti da postoji prazno polje između reči operator i tipa u koji vršimo konverziju (cast).

2) Operator kastovanja nema povratni tip. C++ pretpostavlja da će korisnik vratiti korektan tip kao rezultat funkcije.

Sada, u našem primeru možemo funkciju PrintInt() pozvati na sledeći način:

```
int main()
{
    Cents cCents(7);
    PrintInt(cCents); // print 7

    return 0;
}
```

Kompajler će prvo primetiti da PrintInt preuzima jedan celobrojni parametar. Zatim će primetiti da cCents nije tipa int. Konačno, on će izvršiti pretragu sa ciljem da vidi da li smo obezbedili način da se tip Cents kovertuje u tip int. Pošto to jesmo odradili, kompajler će pozvati našu funkciju operator int(), koja vraća int kao rezultat, a vraćena celobrojna vrednost će biti prosleđena funkciji PrintInt().

Sada je moguće izvršiti i eksplicitno kastovanje promenljive tipa Cents u tip int:

```
Cents cCents(7);
int nCents = static_cast<int>(cCents);
```

PRIMENA PREKLAPANJA KONVERZIJE TIPA

Pri definisanju funkcije preklapanja konverzije tipa neophodno je da postoji prazno polje između reči operator i tipa u koji vršimo konverziju (cast).

Preklapanje operatora kastovanja može biti izvršeno za bilo koji tip podatka, uključujući i korisničke tipove podataka. U nastavku je data nova klasa pod nazivom Dollars koja sadrži operator kastovanja u tip Cents:

```
class Dollars
{
private:
            int m_nDollars;
public:
            Dollars(int nDollars=0)
            {
                        m_nDollars = nDollars;
            }

            // Allow us to convert Dollars into
Cents
            operator Cents() { return
Cents(m_nDollars * 100); }
};
```

Ovo vam omogućava da direktno konvertujete objekat klase Dollars u objekat klase Cents, pa možete uraditi nešto slično sledećem kodu:

```
void PrintCents(Cents cCents)
{
            cout << cCents.GetCents();
}

int main()
{
            Dollars cDollars(9);
            PrintCents(cDollars); // cDollars will
be cast to a Cents

            return 0;
}
```

Stoga će kao rezultat biti ispisana vrednost:

```
90
```

što ima smisla jer 9 dolara ustvari iznosi 900 centi!

Vežbe – Skrivanje podataka, Statički članovi klase

enkapsulacija, sakrivanje podataka, statički članovi klase

- *Studija slučaja - Klasa Time – Pristupne funkcije*
- *Primer - Statički članovi klase*

12

PRISTUPNE I LOKALNE FUNKCIJE - DEKLARACIJA KLASE SALESPERSON

Lokalna funkcija je privatni deo klase koja omogućava izvršavanje operacija neophodnih za javne funkcije klase. Definicija klase ovog primera je smeštena u fajlu SalesPerson.h

Kao što je već spomenuto pristupne funkcije mogu da čitaju i prikazuju podatke. Još jedan dosta uobičajen način korišćenja pristupnih funkcija je u cilju ispitivanja tačnosti uslova, i takve funkcije se nazivaju iskazne (predicate) funkcije.

Primer u nastavku demonstrira korišćenje lokalnih (utility, ili helper - pomoćnih) funkcija. Lokalna funkcija nije deo javnog dela klase. Ona je najčešće privatni deo klase koja omogućava izvršavanje operacija neophodnih za javne funkcije klase. Lokalne funkcije nisu kreirane da bi bile korišćenje od klijenata klase (ali mogu biti korišćenje od strane prijatelja klase). Posmatrajmo sledeću definiciju klase SalesPerson:

```
// SalesPerson.h
// SalesPerson class definition.
// Member functions defined in SalesPerson.cpp.
#ifndef SALESP_H
#define SALESP_H

class SalesPerson
{
public:
   SalesPerson(); // constructor
   void getSalesFromUser(); // input sales from keyboard
   void setSales( int, double ); // set sales for a specific month
   void printAnnualSales(); // summarize and print sales
private:
   double totalAnnualSales(); // prototype for utility function
   double sales[ 12 ]; // 12 monthly sales figures
}; // end class SalesPerson
```

U klasi SalesPerson se deklariše niz od 12 članova koji se odnosi na vrednost prodaje svakog od meseca, i deklarišu se prototipovi konstruktora i metoda koje će da služe za modifikovanje podataka.

PRISTUPNE I LOKALNE FUNKCIJE – DEFINICIJA METODA KLASE SALESPERSON

U cilju ostvarenja koncepta sakrivanja podataka i formiranja interfejsa klase, implementacije deklarisanih metoda klase su smeštene u fajlu SalesPerson.cpp.

Implementacije deklarisanih metoda su smeštene u fajlu SalesPerson.cpp. Kod je dat u nastavku. Kao što možemo videti iz koda konstruktor klase SalesPerson inicijalizuje na 0 sve članove niza sales. Javna funkcija setSales postavlja vrednost prodaje odgovarajućeg meseca, tj člana niza sales. Javna funkcija printAnnualSales štampa ukupnu vrednost prodaje za svaki od 12 meseci. Lokalna (utility) funkcija totalAnnualSales sabira vrednost prodaje po mesecima i dobijamo godišnju vrednost koju zatim štampamo u okviru funkcije printAnnualSales.

```
// SalesPerson.cpp
// Member functions for class SalesPerson.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;
```

```
#include "SalesPerson.h" // include SalesPerson class
definition

// initialize elements of array sales to 0.0
SalesPerson::SalesPerson()
{
   for ( int i = 0; i < 12; i++ )
      sales[ i ] = 0.0;
} // end SalesPerson constructor

// get 12 sales figures from the user at the keyboard
void SalesPerson::getSalesFromUser()
{
   double salesFigure;

   for ( int i = 1; i <= 12; i++ )
   {
      cout << "Enter sales amount for month " << i << ": ";
      cin >> salesFigure;
      setSales( i, salesFigure );
   } // end for
} // end function getSalesFromUser

// set one of the 12 monthly sales figures; function
subtracts
// one from month value for proper subscript in sales array
void SalesPerson::setSales( int month, double amount )
{
   // test for valid month and amount values
   if ( month >= 1 && month <= 12 && amount > 0 )
      sales[ month - 1 ] = amount; // adjust for subscripts
0-11
   else // invalid month or amount value
      cout << "Invalid month or sales figure" << endl;
} // end function setSales
```

PRISTUPNE I LOKALNE FUNKCIJE – DEFINICIJA METODA KLASE SALESPERSON

U cilju ostvarenja koncepta sakrivanja podataka i formiranja interfejsa klase, implementacije deklarisanih metoda klase su smeštene u fajlu SalesPerson.cpp.

```
// print total annual sales (with the help of
utility function)
void SalesPerson::printAnnualSales()
{
   cout << setprecision( 2 ) << fixed
      << "\nThe total annual sales are: $"
      << totalAnnualSales() << endl; // call
utility function
} // end function printAnnualSales

// private utility function to total annual
sales
double SalesPerson::totalAnnualSales()
{
   double total = 0.0; // initialize total

   for ( int i = 0; i < 12; i++ ) // summarize
sales results
      total += sales[ i ]; // add month i sales
to total

   return total;
} // end function totalAnnualSales
```

PRISTUPNE I LOKALNE FUNKCIJE – POKRETAČKA MAIN FUNKCIJA

Korišćenjem main funkcije koja je takođe smeštena u zasebnom main.cpp fajlu ispitujemo funkcionalnost kreirane klase

U nastavku je dat kod za pokretačku main funkciju koja služi za demonstraciju upotrebe lokalnih funkcija. Kao što možemo primetiti učitavanje podataka o vrednostima po mesecima se vrši korišćenjem funkcije članice getSalesFromUser, što znači i da je sama logika manipulacije nad nizom sales enkapsulirana unutar funkcija članica klase Salesperson.

```
// main.cpp
// Demonstrating a utility function.
// Compile this program with SalesPerson.cpp

// include SalesPerson class definition from
SalesPerson.h
#include "SalesPerson.h"

int main()
{
   SalesPerson s; // create SalesPerson object s

   s.getSalesFromUser(); // note simple sequential
code;
   s.printAnnualSales(); // no control statements in
main
   return 0;
} // end main
```

Mogući ulaz i izlaz za prethodni program bi bio:

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```

Studija slučaja - Klasa Time – Pristupne funkcije

sakrivanje, funkcije pristupa, konstruktori

- *Klasa Time – Pristupne funkcije i konstruktor*
- *Time.cpp - Definicija funkcija članica*
- *Main.cpp - Pokretačka main funkcija*
- *Klasa Time – Greške pri kreiranju funkcija pristupa*
- *Klasa Time – Greške pri kreiranju funkcija pristupa - demonstracija primera*

12

KLASA TIME – PRISTUPNE FUNKCIJE I KONSTRUKTOR

U ovom primeru proširujemo klasu Time u cilju demonstracije rada pristupnih funkcija, kao i implicitnog prosleđivanja argumenata konstruktoru.

Kao i bilo koja druga funkcija, konstruktor može da ima podrazumevajuće parametre, kao što je pokazano u narednom primeru, gde imamo 0 kao podrazumevajuću vrednost svih argumenata. U klasi su deklarisane geter i seter metode preko kojih ćemo da pristupamo odgovarajućim privatnim podacima klase. U nastavku je dat prikaz fajla Time.h u kome je deklarisana klasa Time.

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

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

// Time abstract data type definition
class Time
{
public:
   Time( int = 0, int = 0, int = 0 ); // default
constructor

   // set functions
   void setTime( int, int, int ); // set hour, minute,
second
   void setHour( int ); // set hour (after validation)
   void setMinute( int ); // set minute (after validation)
   void setSecond( int ); // set second (after validation)

   // get functions
   int getHour(); // return hour
   int getMinute(); // return minute
   int getSecond(); // return second

   void printUniversal(); // output time in universal-time
format
   void printStandard(); // output 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
```

TIME.CPP - DEFINICIJA FUNKCIJA ČLANICA

Podrazumevajući argumenti konstruktora obezbeđuju da se inicijalizuju vrednosti polja objekta ukoliko u pozivu konstruktora nije prosleđen nijedan stvarni parametar

Kao što se može primetiti iz narednog koda, imamo novu verziju konstruktora koji prima parametre hr, min i sec, koji će da služe za inicijalizaciju privatnih podataka hour, minute i second, respektivno. Konstruktor klase Time poziva funkciju setTime, koja zatim redom poziva funkcije setHour, setMinute i setSecond gde se proverava validnost unešenih podataka, a zatim i postavljanje validnih vrednosti podacima objekta. Podrazumevajući argumenti konstruktora obezbeđuju da se inicijalizuju vrednosti polja objekta ukoliko u pozivu konstruktora nije prosleđen nijedan stvarni parametar. C++ dozvoljava postojanje samo jednog podrazumavajućeg konstruktora u nekoj klasi.

```
// 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 that Time objects start in a consistent state
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec ); // validate and set time
} // 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 )
{
   setHour( h ); // set private field hour
   setMinute( m ); // set private field minute
   setSecond( s ); // set private field second
} // end function setTime

// set hour value
void Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
} // end function setHour
```

MAIN.CPP - POKRETAČKA MAIN FUNKCIJA

Korišćenjem pokretačke main funkcije vršimo testiranje kreirane funkcionalnosti klase Time

U nastavku je data pokretačka funkcija main, smeštena u fajlu main.cpp

```
// main.cpp
// Demonstrating a default constructor for class Time.
#include <iostream>
using std::cout;
using std::endl;

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

int main()
{
   Time t1; // all arguments defaulted
   Time t2( 2 ); // hour specified; minute and second
defaulted
   Time t3( 21, 34 ); // hour and minute specified; second
defaulted
   Time t4( 12, 25, 42 ); // hour, minute and second
specified
   Time t5( 27, 74, 99 ); // all bad values specified

   cout << "Constructed with:\n\nt1: all arguments
defaulted\n  ";
   t1.printUniversal(); // 00:00:00
   cout << "\n  ";
   t1.printStandard(); // 12:00:00 AM

   cout << "\n\nt2: hour specified; minute and second
defaulted\n  ";
   t2.printUniversal(); // 02:00:00
   cout << "\n  ";
   t2.printStandard(); // 2:00:00 AM

   cout << "\n\nt3: hour and minute specified; second
defaulted\n  ";
   t3.printUniversal(); // 21:34:00
}
```

U main funkciji se inicijalizuju 5 objekata klase Time korišćenjem različitih načina prosleđivanja parametara konstruktorima. Prvi objekat će biti inicijalizovan sa sva tri podrazumevajuća argumenta, drugi sa samo jednim specificiranim argumentom (hour), treći sa prva dva specificirana argumenta (hour i minute), četvrti sa specificirana sva tri argumenta, dok će peti imati sve tri pogrešno postavljene vrednosti. Izlaz programa biće:

```
Constructed with:

t1: all arguments defaulted
  00:00:00
  12:00:00 AM

t2: hour specified; minute and second defaulted
  02:00:00
  2:00:00 AM

t3: hour and minute specified; second defaulted
  21:34:00
  9:34:00 PM

t4: hour, minute and second specified
  12:25:42
  12:25:42 PM

t5: all invalid values specified
  00:00:00
  12:00:00 AM
```

KLASA TIME – GREŠKE PRI KREIRANJU FUNKCIJA PRISTUPA

Najčešća greška koja može da se javi pri kreiranju funkcije pristupa je ta da njena vrednost (rezultat) bude referenca na privatni podatak klase

U sledećem primeru je dat uprošćeni isečak koda klase Time koji demonstrira poziv funkcije članice badSetHour koja kao rezultat vraća referencu na privatni podatak klase hour. Na ovaj način direktno pristupamo privatnom članu tako da klijenti klase dobijaju mogućnost da po želji direktno manipulišu privatnim podatkom klase. Primetimo da će se isti problem javiti ako je rezultat funkcije pokazivač na privatni podatak klase.

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

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

class Time
{
public:
   Time( int = 0, int = 0, int = 0 );
   void setTime( int, int, int );
   int getHour();
   int &badSetHour( int ); // DANGEROUS reference
return
private:
   int hour;
   int minute;
   int second;
}; // end class Time
```

Definicija klase Time je smeštena u fajlu Time.cpp čiji je kod naveden u nastavku:

```
// Time.cpp
// Member-function definitions for Time class.
#include "Time.h" // include definition of class Time

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec );
} // end Time constructor

// set values of hour, minute and second
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

// return hour value
int Time::getHour()
{
   return hour;
} // end function getHour

// POOR PROGRAMMING PRACTICE:
// Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
   hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
   return hour; // DANGEROUS reference return
} // end function badSetHour
```

KLASA TIME – GREŠKE PRI KREIRANJU FUNKCIJA PRISTUPA - DEMONSTRACIJA PRIMERA

Pogrešnim definisanjem funkcije pristupa u ovom slučaju se narušava enkapsulacija klase, tj. pretpostavka da korisnici klase ne mogu da imaju direktan pristup privatnim podacima

U cilju demonstracije rada sa pogrešno definisanom funkcijom koristimo pokretačku main funkciju čiji je kod dat u nastavku.

```
// main.cpp
// Demonstrating a public member function that
// returns a reference to a private data member.
#include <iostream>
using std::cout;
using std::endl;

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

int main()
{
   Time t; // create Time object

   // initialize hourRef with the reference returned by
badSetHour
   int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour

   cout << "Valid hour before modification: " << hourRef;
   hourRef = 30; // use hourRef to set invalid value in
Time object t
   cout << "\nInvalid hour after modification: " <<
t.getHour();

   // Dangerous: Function call that returns
   // a reference can be used as an lvalue!
   t.badSetHour( 12 ) = 74; // assign another invalid value
to hour

   cout <<
"\n\n**********************************************\n"
      << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
      << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
      << t.getHour() << endl;
   cout <<
"\n\n**********************************************\n“;
}
```

U okviru main funkcije se prvo deklariše objekat t klase Time, kao i referenca hourref koja se inicijalizuje referencom koja je povratna vrednost funkcije t.badSetHour(20). Na ovaj način se narušava enkapsulacija klase u okviru main funkcije, tj narušava se pretpostavka da korisnici klase ne mogu da imaju direktan pristup privatnim podacima. Izlaz programa biće:

```
Valid hour before modification: 20
Invalid hour after modification: 30

**********************************************
POOR PROGRAMMING PRACTICE!!!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
**********************************************
```

Primer - Statički članovi klase

statički podaci, statičke funkcije, static

- *Definicija klase koja sadrži statičke podatke i funkcije – Fajl Employee.h*
- *Implementacija metoda članica klase – Fajl Employee.cpp*
- *Implementacija metoda članica klase – Fajl Employee.cpp*
- *Testiranje funkcionalnosti implementiranih metoda klase*

12

DEFINICIJA KLASE KOJA SADRŽI STATIČKE PODATKE I FUNKCIJE – FAJL EMPLOYEE.H

Kada se član klase deklariše kao static to znači da bez obzira koliko se bude kreiralo objekata te klase, kreiraće se samo jedna kopija statičkog člana

U narednom primeru pokazujemo upotrebu privatnog statičkog (private static) podatka count kao i javne statičke (public static) funkcije getCount, u okviru klase Employee. Za početak vršimo deklaraciju klase Employee u fajlu Employee.h na sledeći način (definišemo statički podatak kako bi pratili broj kreiranih objekata klase Employee u memoriji):

```
// Employee.h
// Employee class definition.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

class Employee
{
public:
   Employee( const char * const, const char * const ); // constructor
   ~Employee(); // destructor
   const char *getFirstName() const; // return first name
   const char *getLastName() const; // return last name

   // static member function
   static int getCount(); // return number of objects instantiated
private:
   char *firstName;
   char *lastName;

   // static data
   static int count; // number of objects instantiated
}; // end class Employee
```

IMPLEMENTACIJA METODA ČLANICA KLASE – FAJL EMPLOYEE.CPP

Ako se ključna reč static pri definisanju funkcije primeni u liniji koja se nalazi u globalnom opsegu fajla, onda će ta definicija biti vidljiva samo u tom fajlu

Deklaracija članica klase Employee je izvršena u fajlu Employee.cpp. Primetimo da se u 14 liniji ovog fajla vrši definicija i inicijalizacija na nulu statičkog podatka count, a zatim se vrši definicija statičke javne funkcije getCount. Primetimo da nijedna od prethodno navedenih linija ne sadrži ključnu reč static. Razlog je dobro poznat, odnosno ako se ključna reč static primeni u liniji koja se nalazi u globalnom opsegu fajla, onda će ta definicija biti vidljiva samo u tom fajlu. Nasuprot tome, statički član klase treba da bude vidljiv u klijentskom kodu koji ima pristup ovom fajlu, tako da ne smemo da ih deklarišemo kao static u.cpp fajlu. Možemo jedino u *.h fajlu da ih deklarišemo kao static. Podatak klase count prebrojava instancirane objekte klase Employee. Primetimo u definiciji konstruktora korišćenje operatora new u cilju dinamičkog alociranja korektne količine memorije za članove firstName i lastName.

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

#include <cstring> // strlen and strcpy prototypes
using std::strlen;
using std::strcpy;

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

// define and initialize static data member at file scope
int Employee::count = 0;

// define static member function that returns number of
// Employee objects instantiated (declared static in
Employee.h)
int Employee::getCount()
{
   return count;
} // end static function getCount

// constructor dynamically allocates space for first and
last name and
// uses strcpy to copy first and last names into the object
Employee::Employee( const char * const first, const char *
const last )
{
   firstName = new char[ strlen( first ) + 1 ];
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   strcpy( lastName, last );
   count++; // increment static count of employees

   cout << "Employee constructor for " << firstName
      << ' ' << lastName << " called." << endl;
} // end Employee constructor
```

IMPLEMENTACIJA METODA ČLANICA KLASE – FAJL EMPLOYEE.CPP

Ako se ključna reč static pri definisanju funkcije primeni u liniji koja se nalazi u globalnom opsegu fajla, onda će ta definicija biti vidljiva samo u tom fajlu

Deklaracija članica klase Employee je izvršena u fajlu Employee.cpp. Primetimo da se u 14 liniji ovog fajla vrši definicija i inicijalizacija na nulu statičkog podatka count, a zatim se vrši definicija statičke javne funkcije getCount. Primetimo da nijedna od prethodno navedenih linija ne sadrži ključnu reč static. Razlog je dobro poznat, odnosno ako se ključna reč static primeni u liniji koja se nalazi u globalnom opsegu fajla, onda će ta definicija biti vidljiva samo u tom fajlu. Nasuprot tome, statički član klase treba da bude vidljiv u klijentskom kodu koji ima pristup ovom fajlu, tako da ne smemo da ih deklarišemo kao static u.cpp fajlu. Možemo jedino u *.h fajlu da ih deklarišemo kao static. Podatak klase count prebrojava instancirane objekte klase Employee. Primetimo u definiciji konstruktora korišćenje operatora new u cilju dinamičkog alociranja korektne količine memorije za članove firstName i lastName.

```
// destructor deallocates dynamically allocated
memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
      << ' ' << lastName << endl;

   delete [] firstName; // release memory
   delete [] lastName; // release memory

   count--; // decrement static count of employees
} // end ~Employee destructor

// return first name of employee
const char *Employee::getFirstName() const
{
   // const before return type prevents client from
modifying
   // private data; client should copy returned
string before
   // destructor deletes storage to prevent
undefined pointer
   return firstName;
} // end function getFirstName

// return last name of employee
const char *Employee::getLastName() const
{
   // const before return type prevents client from
modifying
   // private data; client should copy returned
string before
   // destructor deletes storage to prevent
undefined pointer
   return lastName;
} // end function getLastName
```

TESTIRANJE FUNKCIONALNOSTI IMPLEMENTIRANIH METODA KLASE EMPLOYEE – FAJL MAIN.CPP

Kada imamo instanciran objekat, onda statička funkcija može biti pozvana i preko imena objekta, a ne samo preko naziva klase

U cilju testiranja statičkih članova koristimo pokretačku main funkciju koja se nalazi u fajlu main.cpp. Kod je dat u nastavku

```
// main.cpp
// Driver to test class Employee.
#include <iostream>
using std::cout;
using std::endl;

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

int main()
{
   // use class name and binary scope resolution operator to
   // access static number function getCount
   cout << "Number of employees before instantiation of any objects is "
      << Employee::getCount() << endl; // use class name

   // use new to dynamically create two new Employees
   // operator new also calls the object's constructor
   Employee *e1Ptr = new Employee( "Susan", "Baker" );
   Employee *e2Ptr = new Employee( "Robert", "Jones" );

   // call getCount on first Employee object
   cout << "Number of employees after objects are instantiated is "
      << e1Ptr->getCount();

   cout << "\n\nEmployee 1: "
      << e1Ptr->getFirstName() << " " << e1Ptr->getLastName()
      << "\nEmployee 2: "
      << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n";

delete e1Ptr; // deallocate memory
   e1Ptr = 0; // disconnect pointer from free-store space
   delete e2Ptr; // deallocate memory
   e2Ptr = 0; // disconnect pointer from free-store space

   // no objects exist, so call static member function getCount again
   // using the class name and the binary scope resolution operator
   cout << "Number of employees after objects are deleted is "
      << Employee::getCount() << endl;
   return 0;
} // end main
```

U main funkciji koristimo statičku metodu članicu getCount da odredimo koliko objekata klase Employee je trenutno instancirano. Primetimo da ako nijedan objekat nije instanciran, onda je moguće pozvati statičku metodu samo korišćenjem Employee::getCount(). Međutim, kada imamo instanciran objekat, onda funkcija getCount može biti pozvana i preko imena objekta. Primetimo da pozivi e2Ptr->getCount() i Employee::getCount() proizvode isti rezultat jer funkcija getCount uvek pristupa istom statičkom članu count. Izlaz programa biće:

```
Number of employees before instantiation of any
objects is 0
 Employee constructor for Susan Baker called.
 Employee constructor for Robert Jones called.
 Number of employees after objects are instantiated
is 2

 Employee 1: Susan Baker
 Employee 2: Robert Jones

 ~Employee() called for Susan Baker
 ~Employee() called for Robert Jones
 Number of employees after objects are deleted is 0
```

Funkcija članica treba biti deklarisana kao static ako ne pristupa nestatičkom podatku ili nestatičkoj funkciji članici klase.

Vežbe – Preklapanje operatora

Preklapanje, operatori, preklopljene funkcije

- *Studija slučaja – Klasa Date*

13

PRIMER: PREKLAPANJE OPERATORA SABIRANJA + ZA RAD SA KOMPLEKSNIM BROJEVIMA

Krajnji dobitak u ovom primeru je da se preklapanjem operatora kompleksni brojevi, kao objekti klase Complex, mogu koristiti na isti način kao i elementarni tipovi podataka

Primer koji sledi pokazuje kako se može preklopiti operator + i u konkretnom slučaju omogućiti veoma jednostavna sintaksa za operacije sa kompleksnim brojevima. Krajnji dobitak je da se kompleksni brojevi, kao objekti klase Complex koriste kao i elementarni tipovi podataka. Lako je razumeti da se na sličan način operator + u matričnom računu može preklopiti tako da se zbir matrica može jednostavno kodirati kao zbir skalara. U ovom slučaju je ostavljena puna sloboda programeru za izmene funkcionalnosti operatora.

```
#include <stdio.h>
class Complex
{
public:
          float real,imag;
          Complex operator + (Complex c);
};

Complex Complex :: operator + (Complex c)
{
          Complex r;
          r.real=real+c.real;
          r.imag=imag+c.imag;
          return r;
}
```

Glavni program u kome ćemo da testiramo funkcionalnost predefinisanog operatora može biti napisan na sledeći način:

```
void main(void)
{
          Complex a,b,c;

          a.real=10;
          a.imag=20;
          b.real=30;
          b.imag=40;

          c=a+b;

          printf(" c=%f+%fj",c.real,c.imag);
}
```

PRIMER. PREKLAPANJE OPERATORA SABIRANJA (+) ZA RAD SA VEKTORIMA

Osnovni oblik preklapanja operatora sabiranja je „Tip operator + (const Tip& a)“

U nastavku je dat primer u kome se vrši preklapanje operatora (+). Prvo ćemo kreirati klasu koja će da služi za čuvanje podataka o dvodimenzionalnom vektoru, a zatim ćemo instancirati dva vektora sa koordinatama: a(3,1) i b(1,2). Operacija sabiranja dva 2D vektora je prosta kao i sabiranje dve x koordinate u cilju dobijanja ukupnog x, odnosno sabiranje dve y koordinate u cilju dobijanja rezultujućeg y. U ovom slučaju imaćemo da je zbir dva vektora jednak (3+1,1+2) = (4,3).

```
// vectors: overloading operators example
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
};

CVector::CVector (int a, int b) {
  x = a;
  y = b;
}

CVector CVector::operator+ (CVector param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}

int main () {
  CVector a (3,1);
  CVector b (1,2);
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0;
}
```

Studija slučaja – Klasa Date

Preklapanje operatora, studija slučaja, klasa Date

- *Definicija klase Date – Datoteka Date.h*
- *Definicija funkcija članica i prijateljskih funkcija – Fajl Date.cpp*
- *Definicija funkcija članica i prijateljskih funkcija – Fajl Date.cpp*
- *Testiranje funkcionalnosti klase Date – Fajl main.cpp*

13

DEFINICIJA KLASE DATE – DATOTEKA DATE.H

Predefinisana (preklopljena) deklaracija je ona deklaracija koja ima isti naziv kao i neka prethodna deklaracija s tim što deklaracije imaju različite argumente i naravno različitu definiciju

Naredni primer demonstrira kreiranje klase Date, koja će naravno da služi za datum. U klasi se vrši preklapanje operatora prefiksnog i postfiksnog inkrementiranja da bi se uveo broj dana u objektu klase Date, što će izazvati prikladno uvećanje broja meseca ili godine ukoliko je neophodno. U fajlu zaglavlja klase Date se specificira javni interfejs klase Date, u kome je deklarisan i preklopljeni operator ubacivanja u tok (stream), podrazumevajući konstruktor, funkcija za postavljanje datuma setDate, preklopljeni prefiksni i postfiksni operator inkrementiranja, preklopljeni kombinovani operator sabiranja i dodele +=, funkcija za testiranje prestupnih godina, i funkcija koja ispituje da li je trenutni dan ustvari poslednji dan u mesecu. U nastavku je dat prikaz fajla zaglavlja Date.h u kome je izvršena definicija klase sa svim preklopljenim operatorima.

```
// Date.h
// Date class definition.
#ifndef DATE_H
#define DATE_H

#include <iostream>
using std::ostream;

class Date
{
   friend ostream &operator<<( ostream &, const Date
& );
public:
   Date( int m = 1, int d = 1, int y = 1900 ); //
default constructor
   void setDate( int, int, int ); // set month, day,
year
   Date &operator++(); // prefix increment operator
   Date operator++( int ); // postfix increment
operator
   const Date &operator+=( int ); // add days,
modify object
   bool leapYear( int ) const; // is date in a leap
year?
   bool endOfMonth( int ) const; // is date at the
end of month?
private:
   int month;
   int day;
   int year;

   static const int days[]; // array of days per
month
   void helpIncrement(); // utility function for
incrementing date
}; // end class Date
```

DEFINICIJA FUNKCIJA ČLANICA I PRIJATELJSKIH FUNKCIJA – FAJL DATE.CPP

Preklopljeni operatori su ustvari funkcije specijalnog naziva koji se sastoji iz ključne reči operator za kojim sledi naziv operatora koji želimo da preklopimo

Definicija funcija članica klase, kao i prijateljskih funkcija je smeštena u fajlu Date.cpp, čiji je kod prikazan u nastavku:

```
// Date.cpp
// Date class member-function definitions.
#include <iostream>
#include "Date.h"

// initialize static member at file scope; one
classwide copy
const int Date::days[] =
   { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31 };

// Date constructor
Date::Date( int m, int d, int y )
{
   setDate( m, d, y );
} // end Date constructor

// set month, day and year
void Date::setDate( int mm, int dd, int yy )
{
   month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
   year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;

   // test for a leap year
   if ( month == 2 && leapYear( year ) )
      day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
   else
      day = ( dd >= 1 && dd <= days[ month ] ) ? dd
: 1;
} // end function setDate
```

U narednom kodu je dat prikaz definicija ostalih metoda klase Date:

```
// overloaded prefix increment operator
Date &Date::operator++()
{
   helpIncrement(); // increment date
   return *this; // reference return to create an
lvalue
} // end function operator++

// overloaded postfix increment operator; note that
the
// dummy integer parameter does not have a parameter
name
Date Date::operator++( int )
{
   Date temp = *this; // hold current state of
object
   helpIncrement();

   // return unincremented, saved, temporary object
   return temp; // value return; not a reference
return
} // end function operator++

// add specified number of days to date
const Date &Date::operator+=( int additionalDays )
{
   for ( int i = 0; i < additionalDays; i++ )
      helpIncrement();

   return *this; // enables cascading
} // end function operator+=
```

DEFINICIJA FUNKCIJA ČLANICA I PRIJATELJSKIH FUNKCIJA – FAJL DATE.CPP

Preklopljeni operatori su ustvari funkcije specijalnog naziva koji se sastoji iz ključne reči operator za kojim sledi naziv operatora koji želimo da preklopimo

Definicija funcija članica klase, kao i prijateljskih funkcija je smeštena u fajlu Date.cpp, čiji je kod prikazan u nastavku:

```
// if the year is a leap year, return true; otherwise, return
false
bool Date::leapYear( int testYear ) const
{
   if ( testYear % 400 == 0 ||
      ( testYear % 100 != 0 && testYear % 4 == 0 ) )
      return true; // a leap year
   else
      return false; // not a leap year
} // end function leapYear

// determine whether the day is the last day of the month
bool Date::endOfMonth( int testDay ) const
{
   if ( month == 2 && leapYear( year ) )
      return testDay == 29; // last day of Feb. in leap year
   else
      return testDay == days[ month ];
} // end function endOfMonth

// function to help increment the date
void Date::helpIncrement()
{
   // day is not end of month
   if ( !endOfMonth( day ) )
      day++; // increment day
   else
      if ( month < 12 ) // day is end of month and month < 12
      {
         month++; // increment month
         day = 1; // first day of new month
      } // end if
```

U narednom kodu je dat prikaz definicija ostalih metoda klase Date:

```
      else // last day of year
      {
         year++; // increment year
         month = 1; // first month of new year
         day = 1; // first day of new month
      } // end else
} // end function helpIncrement

// overloaded output operator
ostream &operator<<( ostream &output, const
Date &d )
{
   static char *monthName[ 13 ] = { "",
"January", "February",
      "March", "April", "May", "June", "July",
"August",
      "September", "October", "November",
"December" };
   output << monthName[ d.month ] << ' ' <<
d.day << ", " << d.year;
   return output; // enables cascading
} // end function operator<<
```

TESTIRANJE FUNKCIONALNOSTI KLASE DATE – FAJL MAIN.CPP

Kod preklapanja operatora treba imati na umu da nije moguće kreirati nove operatore; moguće je samo preklopiti već postojeće operatore.

U cilju testiranja funkcionalnosti klase Date kreiraćemo glavni program i main funkciju u okviru fajla main.cpp:

```
// main.cpp
// Date class test program.
#include <iostream>
using std::cout;
using std::endl;

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

int main()
{
   Date d1; // defaults to January 1, 1900
   Date d2( 12, 27, 1992 ); // December 27, 1992
   Date d3( 0, 99, 8045 ); // invalid date

   cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is "
<< d3;
   cout << "\n\nd2 += 7 is " << ( d2 += 7 );

   d3.setDate( 2, 28, 1992 );
   cout << "\n\n  d3 is " << d3;
   cout << "\n++d3 is " << ++d3 << " (leap year allows
29th)";

   Date d4( 7, 13, 2002 );

   cout << "\n\nTesting the prefix increment operator:\n"
      << "  d4 is " << d4 << endl;
   cout << "++d4 is " << ++d4 << endl;
   cout << "  d4 is " << d4;

   cout << "\n\nTesting the postfix increment operator:\n"
      << "  d4 is " << d4 << endl;
   cout << "d4++ is " << d4++ << endl;
   cout << "  d4 is " << d4 << endl;
   return 0;
} // end main
```

U funkciji main se kreiraju tri objekta tipa Date: objekat d1 se inicijalizuje datumom 1. Januar 1900. godine, d2 se inicijalizuje datumom 27 decembar 1992. godine, a d3 se inicijalizuje nekim nepostojećim datumom. Konstruktor klase Date poziva funkciju setDate u kojoj se ispituje validnost unetih podataka za mesec, dan i godinu. Ukoliko se unese nepostojeći mesec onda se on setuje na 1, nepostojeća godina na 1900, a nepostojeći dan na 1.

Izlaz programa biće:

```
 d1 is January 1, 1900
 d2 is December 27, 1992
 d3 is January 1, 1900

 d2 += 7 is January 3, 1993

   d3 is February 28, 1992
 ++d3 is February 29, 1992 (leap year allows
29th)

 Testing the prefix increment operator:
   d4 is July 13, 2002
 ++d4 is July 14, 2002
   d4 is July 14, 2002

 Testing the postfix increment operator:
   d4 is July 14, 2002
 d4++ is July 14, 2002
```

Zadaci za samostalan rad

Sakrivanje podataka, statički članovi, preklapanje operatora

- *Zadaci za samostalno vežbanje*

14

ZADACI ZA SAMOSTALNO VEŽBANJE

Uz pomoć materijala sa predavanja i vežbi za ovu nedelju, uraditi samostalno sledeće zadatke:

1. Za klasu kvadrat iz prethodne lekcije uraditi sledeće:

Kreirati statički atribut ukupanBrojKvadrata, tipa int, tako da se ovaj atribut uvećava prilikom svakog kreiranja nove instance klase Kvadrat. Kreirati statičku metodu slučajanKvadrat() koja vraća kvadrat sa slučajnim vrednostima veličine stranice, boje ivice i boje unutrašnjosti pri čemu treba da važi:

- veličina stranice je u opsegu [1, 10]
- boja ivice je neka od sledećih vrednosti: "Bela", "Crvena", "Zelena", "Plava", "Crna"
- boja unutrašnjosti je neka od sledećih vrednosti: "Bela", "Crvena", "Zelena", "Plava", "Crna„

2. Napraviti klasu Lekar koja ima od podataka Ime, Prezime i broj dozvole za rad. Zatim napraviti pomoćnu klasu koja ima metodu koja vraća reč od 12 random slova pri čemu je prvo slovo veliko a ostala su mala, i metodu koja vraća slučajni ceo broj u intervalu od 1 do 1200. U main funkciji kreirati dve instance klase Lekar i svaku od njih napuniti random podacima koristeći pomoćnu klasu. Prikazati rezultat rada.

3. Jednu reku naseljavaju ribe, pastrmke i somovi. Napraviti program za izračunavanje procenta somova ako kao parametar imamo ukupan broj riba kao i broj pastrmki. Prilikom pravljenja ove aplikacije treba kreirati klasu koja predstavlja reku (i sve što se tiče riba) i glavnu main funkciju koja je zadužena za sakupljanje podataka i za prikaz podataka.

4. Opisati klasu za rad sa kompeleksnim brojevima. Obezbediti realizaciju operacija sabiranja, oduzimanja množenja, deljenja preklapanjem operatora +,-,*,/. Ispis kompleksnog broja ostvariti preklapanjem operatora <<.

Izdvojiti u fajlove:

- Complex.h – specifikaciju klase i inline funkcije – preklapanje operatora +,-,*,/., konstruktor i destruktor.
- Complex.cpp – realizaciju metoda klase (funkciju za preklapanje insertor operatora)
- Main.cpp – program koji testira klasu.

5. Za klasu kojom se opisuje tačka i sadrži operaciju translacije tačke opisati prijateljske operatorske funkcije koje preklapaju operatore >> i << obezbedjujući formatirano učitavanje i ispis tačke.

Zaključak

16

REZIME

Na osnovu svega obrađenog možemo zaključiti sledeće

Enkapsulacijom se korisnicima klase ograničava direktan pristup njenim skrivenim delovima, da bi se smanjila mogućnost da se poljima objekata dodele pogrešne vrednosti.

Funkcije pristupa su kratke funkcije članice klase, označene kao javne (public), čija je svrha da pristupe vrednosti privatnog člana klase. Pri kreiranju naziva funkcije pristupa obično se uključi naziv promenljive, s tim što prvo slovo postaje veliko a u ime funkcije se ubaci na početku 'set' ili 'get', respektivno.

Prijateljske funkcije u C++-u su specijalna vrsta funkcija koje imaju pristup privatnim i zaštićenim članovima klase. Osim prijateljskih funkcija, neka klasa takođe može biti prijateljska nekoj drugoj klasi. Kada kreiramo prijateljsku klasu onda sve njene funkcije članice postaju prijateljske funkcije te druge klase.

Ponekad, neki isti podatak je potreban za sve članove klase. Kada se član klase deklariše kao static to znači da bez obzira koliko se bude kreiralo objekata te klase, kreiraće se samo jedna kopija statičkog člana. Inicijalizacija statičkog člana se vrši izvan definicije klase korišćenjem operatora pristupa **::** uz ime klase kako bi znali kojoj klasi odgovarajući podatak pripada. Osim podataka mogu postojati i statičke funkcije klase. Statičke funkcije klase su one funkcije koje su nezavisne od objekata klase, i koje mogu biti pozvane iako nije deklarisan nijedan objekat te klase.

Postoje slučajevi u kojima nam je neka promenljiva potrebna samo privremeno i tada koristimo takozvane anonimne promenljive. Anonimna promenljiva je ona promenljiva kojoj nije dodeljeno ime. Anonimne promenljive imaju takozvani "expression scope", što znači da se uništavaju na kraju izraza u kome se kreiraju.

C++ dozvoljava definisanje više od jedne funkcije istog naziva kao i više operatora, što se naziva preklapanje funkcija odnosno preklapanje operatora, respektivno. Predefinisana (preklopljena) deklaracija je ona deklaracija koja ima isti naziv kao i neka prethodna deklaracija (pričamo ovde o istom opsegu važenja), s tim što deklaracije imaju različite argumente i naravno različitu definiciju (implementaciju). U C++-u je moguće predefinisati, tj. preklopiti najveći deo standardnih ugrađenih (build-in) tipova operatora. Na taj način korisnik može da koristi preklopljene operatore nad novim tipovima koje definiše.