# Lekcija 10

# Pokazivači i klase, Nasleđivanje, Kompozicija

*Miljan Milošević*

# POKAZIVAČI I OBJEKTI, NASLEĐIVANJE, KOMPOZICIJA

| | 01 | 02 | 03 | 04 |
|---|---|---|---|---|
| **Uvod** | **Pokazivači i objekti** | **Uvod u nasleđivanje** | **Upotreba izvedenih klasa** | **Dodavanje i menjanje članova izvedene klase** |
| | ➢ *Pokazivači na C++ klase*<br>➢ *Dinamička alokacija objekata korišćenjem operatora new i delete*<br>➢ *Preklapanje operatora pristupa članovima klase (->)*<br>➢ *Primer preklapanja operatora pristupa članovima klase (->)*<br>➢ *Primer preklapanja operatora pristupa članovima klase (->)* | ➢ *Osnovna razmatranja*<br>➢ *Osnovne i izvedene klase*<br>➢ *Primer izvođenja novih klasa*<br>➢ *Nasleđivanje i kontrola pristupa*<br>➢ *Tipovi nasleđivanja*<br>➢ *Primer javnog nasleđivanja* | ➢ *Zaštićeni članovi bazne klase*<br>➢ *Kreiranje izvedenih klasa*<br>➢ *Korišćenje izvedenih klasa*<br>➢ *Predefinisani (overloaded) konstruktor*<br>➢ *Upotreba predefinisanog (overloaded) konstruktora* | ➢ *Uvodna razmatranja*<br>➢ *Dodavanje nove funkcionalnosti*<br>➢ *Nadglasavanje funkcija (overriding)*<br>➢ *Upotreba nadglasavajućih i nadglasanih funkcija*<br>➢ *Dodavanje nove funkcionalnosti postojećoj funkciji* |

# POKAZIVAČI I OBJEKTI, NASLEĐIVANJE, KOMPOZICIJA

## 05 Sakrivanje članova u izvedenoj klasi

- *Sakrivanje metoda bazne klase*
- *Sakrivanje podataka bazne klase*

## 06 Višestruko nasleđivanje

- *Uvodna razmatranja*
- *Uvodna razmatranja*
- *Problemi kod višestrukog nasleđivanja*
- *Problem dijamanta kod višestrukog nasleđivanja*

## 07 Kompozicija

- *Uvod u kompoziciju*
- *Upotreba klasa u drugim klasama – Korišćenje liste za inicijalizaciju*
- *Kompletan primer kompozicije – Klasa Point2D*
- *Kompletan primer kompozicije – Klasa Creature*
- *Kompletan primer kompozicije – Glavni program*
- *Zašto koristiti kompoziciju?*

## 08 Agregacija

- *Uvod u agregaciju*
- *Primer agregacije*
- *Razlike između kompozicije i agregacije*

## 09 Vežbe

- ❑ *Zadaci za samostalan rad*

# UVOD

## *Ova lekcija treba da ostvari sledeće ciljeve:*

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

- Pokazivači i objekti
- Nasleđivanje
- Kompozicija i agregacija

Objektima neke klase je moguće pristupiti korišćenjem pokazivača. Jednom deklarisana klasa postaje validni tip podatka na koji možemo usmeriti pokazivač. Da bi se pristupilo članu klase korišćenjem pokazivača na klasu, koristi se operator pristupa **->** na isti način kako se ovaj operator koristi kod pokazivača na strukturu.

Nasleđivanje je jedna od najvažnijih osobina objektno orijentisanog programiranja. Ono omogućava kreiranje vrlo složenih klasa krećući se od osnovih klasa, što olakšava kreiranje i održavanje aplikacije. Pri izradi programa, umesto da se kompletno od nule pišu podaci članovi i funkcije članice, programer ima mogućnost da navede da će nova klasa naslediti osobine postojeće klase.

Programeri imaju mogućnost da naslede funkcionalnost bazne klase, dodaju novu funkcionalnost, modifikuju postojeću funkcionalnost ili da sakriju funkcionalnost koja nije potrebna.

U C++-u klasa može biti izvedena iz više od jedne klase, što znači da može naslediti podatke i metode iz više baznih klasa istovremeno. Ovakvo nasleđivanje se naziva višestruko nasleđiivanje, a njegove prednosti i mane će biti opisane u okviru ove lekcije.

U stvarnom životu, složeni objekti se često grade od manjih, jednostavnijih objekata. Da bi se olakšala izgradnja složenih klasa iz jednostavnijih, C ++ nam omogućava da obavljamo kompoziciju objekata na vrlo jednostavan način - pomoću klase kao promenljive članice u drugim klasama. Kompozicija stoga označava upotrebu jedne ili više klasa u okviru definicije tj. deklaracije druge klase. Specijalan slučaj kompozicije je agregacija u kojoj se ne podrazumeva vlasništvo kompleksnog objekta nad njegovim podobjektima.

# Pokazivači i objekti

*pokazivači, objekti, dinamički objekti, preklapanje operatora*

- *Pokazivači na C++ klase*
- *Dinamička alokacija objekata korišćenjem operatora new i delete*
- *Preklapanje operatora pristupa članovima klase (->)*
- *Primer preklapanja operatora pristupa članovima klase (->)*

01

# POKAZIVAČI NA C++ KLASE

*Objektima neke klase je moguće pristupiti korišćenjem pokazivača. Da bi se pristupilo članu klase korišćenjem pokazivača na klasu, koristi se operator pristupa ->*

Objektima neke klase je moguće pristupiti korišćenjem pokazivača. Jednom deklarisana, klasa postaje validni tip podatka na koji možemo usmeriti pokazivač. Deklarisanje pokazivača na klasu može biti izvršeno na sledeći način:

```
Rectangle * prect;
```

gde će prect biti pokazivač na objekat klase Rectangle. Pokazivač na klase se koristi na isti način kao i pokazivač na strukture. Posmatrajmo sledeći primer kako bi bolje razumeli koncept pokazivača na klase:

```
#include <iostream>
using namespace std;
class Box
{
   public:
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
      }
      double Volume()  { return length * breadth *
height;}
   private:
      double length;  double breadth; double height;
};
```

Da bi se pristupilo članu klase korišćenjem pokazivača na klasu, koristi se operator pristupa **->** na isti način kako se ovaj operator koristi kod pokazivača na strukturu. Takođe, kao pri radu sa svim ostalim pokazivačima, neophodno je inicijalizovati pokazivač pre samog korišćenja. U nastavku je main funkcija kojom ćemo da testiramo klasu Box:

```
int main(void)
{
   Box Box1(3.3, 1.2, 1.5);    // Declare box1
   Box Box2(8.5, 6.0, 2.0);    // Declare box2
   Box *ptrBox;                // Declare pointer to
a class.

   ptrBox = &Box1;

   cout << "Volume of Box1: " << ptrBox->Volume() <<
endl;

   ptrBox = &Box2;

   cout << "Volume of Box2: " << ptrBox->Volume() <<
endl;

   return 0;
}
```

Nakon izvršavanja programa dobiće se sledeći rezultat:

```
Constructor called.
Constructor called.
Volume of Box1: 5.94
Volume of Box2: 102
```

# DINAMIČKA ALOKACIJA OBJEKATA KORIŠĆENJEM OPERATORA NEW I DELETE

*Kao i kod ostalih tipova podataka, operatore new i delete možemo koristiti za dinamičko alociranje memorije neophodne za čuvanje objekta, odnosno oslobađanje, respektivno*

Operatore new i delete možemo koristiti na isti način i kod dinamičkog alociranja memorije neophodnog za kreiranje objekata. Uzmimo u obzir sledeću deklaraciju i iskaz:

```
Time *timePtr;
timePtr = new Time;
```

Operator new alocira odgovarajući memorijski prostor neophodan za smeštanje objekta tipa Time, poziva podrazumevajući konstruktor koji inicijalizuje podatke objekta i kao rezultat vraća pokazivač na tip podatka koji se nalazi sa desne strane operatora new (tj. tip je Time *). Ukoliko operator new nije u stanju da odvoji odgovarajući memorijski prostor nastaće greška, i izbaciće se izuzetak ("throwing an exception."). Da bi se oslobodio memorijski prostor zauzet korišćenjem operatora new, koristi se operator delete:

```
delete timePtr;
```

U nastavku je dat primer koji demonstrira korišćenje pokazivača na objekte i korišćenje dinamičkog alociranja memorije:

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

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

Prethodnu klasu ćemo testirati u sledećem glavnom programu:

```
int main() {
  Rectangle obj (3, 4);
  Rectangle * foo, * bar, * baz;
  foo = &obj;
  bar = new Rectangle (5, 6);
  baz = new Rectangle[2] { {2,5}, {3,6} };
  cout << "obj's area: " << obj.area() << '\n';
  cout << "*foo's area: " << foo->area() << '\n';
  cout << "*bar's area: " << bar->area() << '\n';
  cout << "baz[0]'s area:" << baz[0].area() << '\n';
  cout << "baz[1]'s area:" << baz[1].area() << '\n';
  delete bar;
  delete[] baz;
  return 0;
}
```

U prethodnom primeru je korišćeno nekoliko operatora nad objektima i pokazivačima (operatori *, &, ., ->, []). Njih možemo interpretirati na sledeći način:

*x - pokazivač na x
&x - adresa promenljive x
x.y - član y objekta x
x->y - član y objekta na koji pokazuje x
(*x).y - član y objekta na koji poakazuje x (isto kao prethodno)
x[0] - prvi objekat na koji pokazuje x
x[1] - drugi objekat na koji pokazuje x
x[n] - (n+1)-vi objekat na koji pokazuje x

# PREKLAPANJE OPERATORA PRISTUPA ČLANOVIMA KLASE (->)

*Operator -> se koristi veoma često u spoju sa pokazivačkim operatorom dereferenciranja * u cilju implementacije takozvanih pametnih pokazivača*

Operator pristupa članu klase (->) može biti preklopljen ali je postupak malo komplikovan. On se definiše sa ciljem da se tipu klase dodeliko takozvano „pokazivačko ponašanje“. Ovaj operator -> mora biti funkcija članica klase. U slučaju da se koristi, povratni tip te funkcije mora biti pokazivač ili objekat klase na koji želimo da primenimo ovaj operator.

Operator -> se koristi veoma često u spoju sa pokazivačkim operatorom dereferenciranja *, u cilju implementacije takozvanih pametnih pokazivača ("smart pointers"). Ovi pokazivači su objekti koji se ponašaju kao i normalni pokazivači s tim da oni obavljaju druge zadatke kada se preko njih pristupi objektu, kao što je na primer automatsko brisanje objekta u trenutku kada se pokazivač uništava ili kada se pokazivač iskoristi za pokazivanje na drugi objekat.

Operator dereferenciranja -> može biti definisan kao unarni postfiksni operator. Tako, u datoj klasi:

```
class Ptr{
   //...
   X * operator->();
};
```

objekat klase **Ptr** može biti korišćen da se pristupi članovima klase X na sličan način kako se koriste i pokazivači. Na primer:

```
void f(Ptr p )
{
   p->m = 10 ; // (p.operator->())->m = 10
}
```

pri čemu se iskaz p->m interpretira kao (p.operator->())->m.

# PRIMER PREKLAPANJA OPERATORA PRISTUPA ČLANOVIMA KLASE (->)

*U slučaju preklapanja operatora dereferenciranja -> povratni tip te preklopljene funkcije mora biti pokazivač ili objekat klase na koji želimo da primenimo ovaj operator*

Korišćenjem istog koncepta naredni primer opisuje kako operator pristupa članu klase -> može biti preklopljen.

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

class Obj {
 static int i, j;
public:
 void f() const { cout << i++ << endl; }
 void g() const { cout << j++ << endl; }
};

int Obj::i = 10;
int Obj::j = 12;

class ObjContainer {
 vector<Obj*> a;
public:
 void add(Obj* obj)
 {
  a.push_back(obj);  // call vector's standard
method.
 }
 friend class SmartPointer;
};
```

Test klasa i glavni program mogu biti napisani na sledeći način:

```
class SmartPointer {
 ObjContainer oc;
 int index;
public:
 SmartPointer(ObjContainer& objc) {
  oc = objc;
  index = 0;
 }
 bool operator++() // Prefix version {
  if(index >= oc.a.size()) return false;
  if(oc.a[++index] == 0) return false;
  return true;
 }
 bool operator++(int)  {
  return operator++();
 }
 Obj* operator->() const  {
  if(!oc.a[index])  {
   cout << "Zero value";
   return (Obj*)0;
  }
  return oc.a[index];
 }
};
```

Rezultat programa su oštampani brojevi od 10 do 21.

# PRIMER PREKLAPANJA OPERATORA PRISTUPA ČLANOVIMA KLASE (->)

*U slučaju preklapanja operatora dereferenciranja -> povratni tip te preklopljene funkcije mora biti pokazivač ili objekat klase na koji želimo da primenimo ovaj operator*

Korišćenjem istog koncepta naredni primer opisuje kako operator pristupa članu klase -> može biti preklopljen.

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

class Obj {
 static int i, j;
public:
 void f() const { cout << i++ << endl; }
 void g() const { cout << j++ << endl; }
};

int Obj::i = 10;
int Obj::j = 12;

class ObjContainer {
 vector<Obj*> a;
public:
 void add(Obj* obj)
 {
  a.push_back(obj);  // call vector's standard
method.
 }
 friend class SmartPointer;
};
```

Test klasa i glavni program mogu biti napisani na sledeći način:

```
int main() {
 const int sz = 10;
 Obj o[sz];
 ObjContainer oc;
 for(int i = 0; i < sz; i++)
 {
  oc.add(&o[i]);
 }
 SmartPointer sp(oc); // Create an iterator
 do {
  sp->f(); // smart pointer call
  sp->g();
 } while(sp++);
 return 0;
}
```

Rezultat programa su oštampani brojevi od 10 do 21.

# Uvod u nasleđivanje

*nasleđivanje, bazna klasa, izvedene klase*

- *Osnovna razmatranja*
- *Osnovne i izvedene klase*
- *Primer izvođenja novih klasa*
- *Nasleđivanje i kontrola pristupa*
- *Tipovi nasleđivanja*
- *Primer javnog nasleđivanja*

**02**

# OSNOVNA RAZMATRANJA

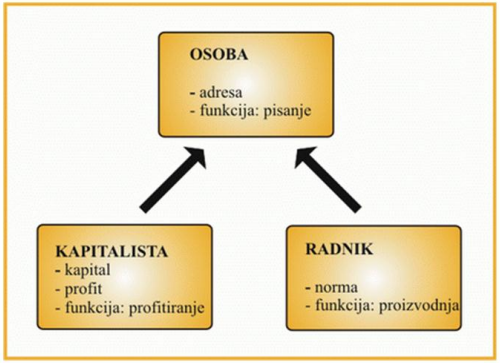
*Nasleđivanje omogućava kreiranje vrlo složenih klasa krećući se od osnovih klasa, što olakšava kreiranje i održavanje aplikacije*

Nasleđivanje je jedna od najvažnijih osobina objektno orijentisanog programiranja. Ono omogućava kreiranje vrlo složenih klasa krećući se od osnovih klasa, što olakšava kreiranje i održavanje aplikacije. Ovo takođe obezbeđuje mogućnost ponovnog korišćenja napisanog koda što ubrzava samo pisanje aplikacije.

Pri izradi programa, umesto da se kompletno od nule pišu podaci članovi i funkcije članice, programer ima mogućnost da navede da će nova klasa naslediti osobine postojeće klase. Postojeća klasa se naziva osnovna ili bazna klasa (base class), dok se nove klase nazivaju izvedene klase (derived class). Koncept nasleđivanja je put ka formiranju organizovanih familija klasa koje mogu dalje da se proširuju. To u mnogome podseća na programerske biblioteke funkcija.

Pođimo od jednog primera definisanja klasa i objekata koji će pokazati neke ideje objektnog sagledavanja problema. Definišimo dve klase: KAPITALISTA i RADNIK. Na primer klasi KAPITALISTA prirodno je pridružiti podatke kapital i profit (brojni tipovi podataka) i funkciju profitiranje. Klasi RADNIK možemo dodeliti podatke plata i norma i funkciju proizvodnja. Kao što se vidi radi se o podacima i funkcijama koji se vezuju za određenu osobu koja se može identifikovati adresom.

Možemo definisati i klasu OSOBA koja sadrži podatak adresa i npr. funkciju pisanje. Klase KAPITALISTA i RADNIK pored već pridruženih podataka i funkcija članica treba da uključe podatke i funkcije sadržane u klasi OSOBA.




Slika-1 Primer nasleđivanja iz bazne klase Osoba

Ovaj jednostavan grafički prikaz pokazuje definicije klasa i pokazuje da klase KAPITALISTA i RADNIK nasleđuju sadržaj klase OSOBA; podrazumevaju sve što i klasa OSOBA sa dodatkom članova specifičnih za svaku od izvedenih klasa.

# OSNOVNE I IZVEDENE KLASE

*Programer ima mogućnost da navede da će nova klasa naslediti osobine postojeće klase. Postojeća klasa se naziva osnovna ili bazna klasa dok se nove klase nazivaju izvedene klase*

U C++-u klasa može biti izvedena iz više od jedne klase, što znači da može naslediti podatke i metode iz više baznih klasa istovremeno. Da bi definisali izvedenu klasu, koristimo listu izvođenja kojom se specificiraju bazne klase. Lista izvođenja klase ima sledeći oblik:

```
class derived-class: access-specifier base-class
```

gde je derived-class ime izvedene klase, access-specifier – specifikator pristupa koji može biti public, protected, ili private, a base-class je ima bazne klase iz koje izvodimo novu klasu. Ukoliko se ne koristi specifikator pristupa podrazumeva se da je private. U nastavku je dat primer nasleđivanja u preduzeću gde kao objekte možemo imati osobu, kapitalistu i radnika. Klasa Osoba može biti napisana na sledeći način:

```
class Osoba
{
public:
          char adresa[20];
          void pisanje(void);
}
```

Specifikator pristupa ograničava najpristupačniji nivo za sve članove izvedene klase: Članice koje imaju veći nivo pristupa se ograničavaju na nivo koji je specificiran, dok članovi istog ili restriktivnijeg nivoa zadržavaju taj restriktivni nivo prilikom nasleđivanja.

Klase Kapitalista i Radnik mogu biti napisane kao:

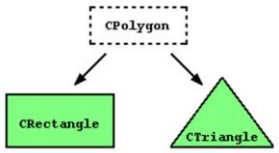
```
class Kapitalista : public Osoba
{
public:
          float profit;
          void profitiranje(float);
          void pisanje(char *poslovni_tekst);
private:
          float kapital;
}

class Radnik : public Osoba
{
public:
          int norma;
          int proizvodnja(int);
}
```

Pri nasleđivanju se podrazumeva nasleđivanje svih članova klase od kojih se izvodi klasa naslednica. Članovi koji su deklarisani unutar tela izvedene klase su novi podaci i funkcije kojima se klasa proširuje u odnosu na onu iz koje je izvedena. Ne mora uvek biti tako. Neke funkcije se mogu predefinisati u izvedenoj klasi. To znači da ostaje isto ime funkcije, a parametri i tip funkcije su uslovljeni novim potrebama. Prototip takve funkcije se mora naći u telu izvedene klase. Ovde treba prepoznati polimorfizam u OOP jer funkcija sa istim imenom radi različite stvari u različitim objektima.

# PRIMER IZVOĐENJA NOVIH KLASA

*Kada jedna klasa nasleđuje drugu klasu, članovi izvedene klase mogu da pristupe zaštićenim članovima osnovne klase, ali nemaju pristup privatnim članovima*

Pretpostavimo da želimo da kreiramo seriju klasa koje će da služe za opisivanje poligona, kao što su npr pravougaonik i trougao (Crectangle i CTriangle). I pravougaonik i trougao imaju zajedničke osobine, tj mogu biti opisani preko dva parametra: visina i osnovica (height i base). Korišćenjem principa o nasleđivanju, prethodne relacije mogu biti opisane uvođenjem klase CPolygon iz koje ćemo izvesti dve klase za trougao i pravouganik: CRectangle i CTriangle.




Slika-2 Primer nasleđivanja

Klasa CPolygon će sadržati podatke članove klji su zajednički za oba tipa poligona, u našem slučaju to su visina i širina ( width i height).

Klase CRectangle i CTriangle će biti njene naslednice (izvedene klase) koje će osim zajedničkih imati i osobine specifične za jedan odnosno drugi tip poligona. U nastavku je dat primer gde kreiramo klase Polygon, Rectangle i Triangle

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

class Polygon {
protected:
            int width, height;
public:
            void set_values (int
a, int b)
            { width=a; height=b;}
};

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

class Triangle: public Polygon {
public:
            int area ()
            { return width *
height / 2; }
};
```

Objekti klasa Rectangle i Triangle sadrže članove koje su nasledili iz klase Polygon, i to su: width, height i set_values. Specifikator pristupa označen kao protected, u klasi Polygon, je sličan specifikatoru private. Jedina razlika nastaje u slučaju nasleđivanja: Kada jedna klasa nasleđuje drugu klasu, članovi izvedene klase mogu da pristupe zaštićenim članovima osnovne klase, ali nemaju pristup privatnim članovima. Deklarisanjem članica bazne klase width i height kao protected umesto private, ovi podaci postaju dostupni i u izvedenim klasama Rectangle i Triangle, a ne samo funkcijama članicama klase of Polygon. U slučaju da su ovi deklarisani kao javni, njima bi se moglo pristupi iz bilo kog dela programa.

# NASLEĐIVANJE I KONTROLA PRISTUPA

*Članove bazne klase treba definisati kao privatne ukoliko želimo da sprečimo pristup od strane funkcija izvedene klase*

Izvedena klasa može da pristupi svim članovima bazne klase koji nisu definisani kao privatni. Stoga, članove bazne klase treba definisati kao privatne ukoliko želimo da sprečimo pristup od strane funkcija izvedene klase. Stoga, možemo da sumiramo koji su to tipovi pristupa odgovarajućim članovima bazne klase na sledeći način:

| Pristup | public | protected | private |
|---|---|---|---|
| Unutar klase | da | da | da |
| Izvedena klasa | da | da | ne |
| Izvan klase | da | ne | ne |

Slika-3 Tipovi pristupa odgovarajućim članovima bazne klase

gde su pod grupom „izvan klase“ sadržani pozivi iz npr. main funkcije, drugih funkcija ili klasa koje nisu u srodstvu sa baznom klasom.

Izvedena klasa nasleđuje sve osnovne metode bazne klase, osim sledećih izuzetaka:

- Konstruktor, destruktor i konstruktor kopiranja bazne klase.
- Predefinisane (preklopljene) operatore osnovne klase.
- Prijateljske funkcije osnovne klase.
- Privatne članove bazne klase.

# TIPOVI NASLEĐIVANJA

*Tip nasleđivanja se definiše korišćenjem specifikatora pristupa koji može biti public, protected ili private. Stoga, nasleđivanje može biti privatno, zaštićeno ili javno*

Kada izvodimo neku klasu iz bazne klase, bazna klasa može biti nasleđenja korišćenjem specifikatora public, protected ili private , što znači da se ostvaruje privatno, zaštićeno ili javno nasleđivanje. Tip nasleđivanja se definiše korišćenjem specifikatora pristupa koji su prethodno opisani. Veoma je radak slučaj da se koristi privatno i zaštićeno nasleđivanje, za razliku od javnog nasleđivanja koje se najčešće koristi. Pri korišćenju ovih pomenuih načina nasleđivanja važe sledeća pravila:

- Javno nasleđivanje: Kada se nasleđivanje vrši iz javne (public) bazne klase, javni (public) članovi bazne klase postaju javni (public) članovi izvedene klase, a zaštićeni (protected) članovi bazne klase postaju zaštićeni (protected ) članovi izvedene klase. Privatni (private) članovi bazne klase nisu dostupni u izvedenoj klasi, ali im se može pristupiti korišćenjem javnih i zaštićenih metoda bazne klase.
- Zaštićeno nasleđivanje: Kada se nasleđivanje vrši iz javne (protected ) bazne klase javni (public) i zaštićeni (protected) članovi bazne klase postaju zaštićeni (protected ) članovi izvedene klase.
- Privatno nasleđivanje: Kada se nasleđivanje vrši iz privatne (private) bazne klase, public i protected članovi bazne klase postaju privatni (private) članovi izvedene klase.

U jednom od prethodnih primera, imali smo nasleđivanje osobina klase Polygon od strane klasa Rectangle and Triangle, gde su nasleđeni članovi zadržali istu dozvolu pristupa koju su imali u osnovnoj klasi Polygon:

```
Polygon::width           // protected access
Rectangle::width         // protected access

Polygon::set_values()    // public access
Rectangle::set_values()  // public access
```

Ovo je ostvareno jer je relacija nasleđivanja definisana korišćenjem ključne reči public uz ime bazne klase:

```
class Rectangle: public Polygon { /* ... */ }
```

Pošto je public najveći nivo pristupa, navođenjem ove ključne reči pri definiciji izvedene klase, obezbedićemo da pristup u izvedenoj klasi bude isti kao i u baznoj klasi. Na primer, ukoliko imamo klasu Daughter koja nasleđuje klasu Mother korišćenjem specifikatora protected:

```
class Daughter: protected Mother;
```

svi javni članovi klase Mother će u izvedenoj klasi postati protected. Naravno, ovo neće uticati na nove članice koje definišemo u okviru klase Doughter, koji mogu biti public. Ukoliko se ne specificira nivo pristupa kompajler će usvojiti da je on private.

# PRIMER JAVNOG NASLEĐIVANJA

*Iako se konstruktor i destruktor bazne klase ne nasleđuju u izvedenoj klase, oni se pozivaju kao prva odnosno poslednja operacija iz konstruktora i destruktora izvedene klase*

Iako se konstruktor i destruktor bazne klase ne nasleđuju u izvedenoj klase, oni se i dalje pozivaju iz konstruktora i destruktora izvedene klase. Osim ako nije drugačije navedeno, konstruktor izvedene klase poziva podrazumevajući konstruktor njihovih baznih klasa, koji mora da postoji.
Moguće je pozvati i drugi konstruktor bazne klase koji nije podrazumevajući (konstruktor koji nema parametre), i to se uglavnom radi korišćenjem liste za inicijalizaciju, na isti način kao kod inizicijalizacije članova klase:

```
derived_constructor (parameters) : base_constructor (parameters) {...}
```

U nastavku je dat primer gde imamo tri klase: Mother, Son i Daughter. Klasa Mother ima sledeći oblik:

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

class Mother
{
public:
            Mother ()
            { cout << "Mother: no parameters\n"; }
            Mother (int a)
            { cout << "Mother: int parameter\n"; }
};
```

Klase Daughter i Son, kao i main funkcija za testiranje mogu biti napisani na sledeći način:

```
class Daughter : public Mother {
public:
            Daughter (int a)
            { cout << "Daughter: int parameter\n\n";
}
};
class Son : public Mother {
public:
            Son (int a) : Mother (a)
            { cout << "Son: int parameter\n\n"; }
};
int main () {
            Daughter kelly(0);
            Son bud(0);

            return 0;
}
```

Primetimo razliku između načina poziva konstruktora klase Mother pri kreiranju objekta tipa Daughter, i načina poziva konstuktora kada se kreira objekat klase Son. Razlike se javljaju kao posledica različite implementacije konstruktora klasa Daughter i Son:

```
Daughter (int a)          // call default constructor
Son (int a) : Mother (a)  // call this specific constructor
```

# Upotreba izvedenih klasa

*nasleđivanje, izvedene klase, overload, konstruktor*

- *Zaštićeni članovi bazne klase*
- *Kreiranje izvedenih klasa*
- *Korišćenje izvedenih klasa*
- *Predefinisani (overloaded) konstruktor*
- *Upotreba predefinisanog (overloaded) konstruktora*

**03**

# ZAŠTIĆENI ČLANOVI BAZNE KLASE

*Generalno gledano, 'protected' se koristi umesto 'private' kad god se zna da će biti uvedene izvedene klase u odnosu na neku baznu klasu*

Članovi bazne klase označeni kao privatni nisu dostupni izvedenoj klasi. Ovi privatni članovi mogu biti deklarisani da su zaštićeni, '*protected*' , što im omogućuje da se automatski nasleđuju u izvedenim klasama. Npr. deklaraciju bazne klase People možemo napisati u fajlu zaglavlja, "people.h":

```
#include <string>
class People
{
public:
          void setPhone(int y) {phone = y;}
          int getPhone() {return phone;}
protected:
          string surname;
          int phone;
};
```

Generalno gledano, '*protected*' se koristi umesto '*private*' kad god se zna da će biti uvedene izvedene klase u odnosu na neku baznu klasu. Nije poželjno deklarisati zaštićene članove da su javni, time omogućavajući direktan pristup iz spoljašnjem sveta. Dakle, ako je klasa Y izvedena iz klase X, javni članovi klase X se nasleđuju kao javni članovi klase Y, zaštićeni članovi klase X se nasleđuju kao zaštićeni članovi klase Y, ali, privatni članovi klase X se ne nasleđuju u klasi Y.

Kad se kaže da se neki članovi klase Y nasleđuju iz klase X, to znači da instrukcije tj. programski kod koji definiše te članove u klasi X (u baznoj klasi) se automatski prepisuju u klasu Y, tj. podrazumeva se da taj programski kod iz klase X postoji i u klasi Y. Dakle, sintaksa je sledeća:

```
class Y: public X { ... };
class X  {
public:
          ...
protected:
          ...
};
```

# KREIRANJE IZVEDENIH KLASA

*Članovi izvedene klase koji su zajednički sa baznom klasom su definisani u baznoj klasi, dok članovi specifični za izvedenu klasu mogu se definisati u izvedenoj klasi*

Članovi izvedene klase koji su zajednički sa baznom klasom su definisani u baznoj klasi, dok članovi specifični za izvedenu klasu mogu se definisati u izvedenoj klasi. Izvedena klasa automatski nasleđuje javne i zaštićene članove bazne klase. Izvedene klase kombinuju nasleđene i izvedene elemente tj. članove. U nastavku je dat primer izvedene klase Friend, čija je deklaracija napisana u fajlu *friend.h*, koji ima sledeći oblik:

```
#include "people.h"
class Friend : public People
{
public:
            void setBirthday(int b) {birthday = b;}
            int getBirthday() {return birthday;}
protected:
            int birthday;
};
```

Klasa Friend je izvedena klasa, ali iz klase Friend se može izvesti neka nova klasa, pa klasa Friend može postati bazna klasa za neku novu izvedenu klasu iz klase Friend. Onda će *protected* članovi ove klase biti dostupni novoj izvedenoj klasi. Pošto je *protected* fleksibilnije od *private*, često se *protected* koristi umesto *private* u slučajevima da neka klasa može biti u budućnosti tretirana kao bazna klasa.

Dakle, reč *protected* se koristi u baznim klasama, ali postoje hijerarhije klasa sa nekoliko nivoa nasleđivanja, pa neka klasa može biti istovremeno i bazna i izvedena.

U nastavku je dat primer klase Macka, u fajlu macka.h, koja koristi baznu klasu Sisar, iz fajla sisar.h:

```
#include "sisar.h"
#include <string>
class Macka : public Sisar
{
public:
 //konstruktor/destruktor
 Macka( string inicIme)
 {
  ime = inicIme;
 }
 ~Macka(){}
 //inline get metoda
 string getIme() const { return name;}
 //inline const metoda
 void mjau() const {cout << “mjau”<<endl;}
protected:
 string ime;
};
```

# KORIŠĆENJE IZVEDENIH KLASA

*Izvedena klasa automatski nasleđuje javne i zaštićene članove bazne klase. Izvedene klase kombinuju nasleđene i izvedene elemente tj. članove*

Izvedene klase omogućuju osobinu nasleđivanja javnih i zaštićenih članova iz bazne klase. U primeru koji sledi prikazana je upotreba izvedenih klasa, gde je kreiran jedan objekat po menu John, klase Friend. Kao što vidimo, koriste se setter/getter funkcije nasleđene iz bazne klase kao i funkcije izvedene klase. Glavni program je kreiran u okviru fajla *friend.cpp*, pri čemu se definicija klase Friend uključuje iz fajla *friend.h*. Primer:

```
#include "friend.h"
int main()
{
            Friend John;
            John.setPhone(1234567);
            John.setBirthday(250386);
            cout<< John.getPhone<< endl;
            cout << John.getBithday<<endl;
            return 0;
}
```

U nastavku je dat još jedan primer gde je glavni program napisan u fajlu macka.cpp koji koristi deklaracije iz fajla macka.h:

```
#include "macka.h"
int main()
{
            Macka MojaMacka("Macka");
            MojaMacka.setStarost(3);
            MojaMacka.setTezina(2);
            MojaMacka.mjau();
            cout << MojaMacka.getIme()<<endl;
            cout << MojaMacka.getStarost()<<endl;
            MojaMacka.getTezina()<<endl;
            return 0;
}
```

# PREDEFINISANI (OVERLOADED) KONSTRUKTOR

*Predefinisani ('overloaded') konstruktori se mogu primeniti da omoguće različite liste argumenata, različit broj ili tip argumenata a mogu se koristiti i kod kreiranja objekata*

Predefinisani ('*overloaded*') konstruktori se mogu primeniti da omoguće različite liste argumenata, različit broj ili tip argumenata; a mogu se koristiti i kod kreiranja objekata. U nastavku je dat primer klase Friend koja nasleđuje klasu People:

```
#include "people.h"
#include <string>
class Friend : public People
{
public:
          Friend(int bd) { birthday = bd;}
          Friend(int bd, string sn) { birthday = bd; surname = sn;}
          ~Friend(){} //destructor
protected:
          int birthday;
};
```

Kod kreiranja objekata, preklopljeni konstruktori se koriste da inicijalizuju različite skupove promenljivih. Primer prekloljenih konstruktora u izvedenoj klasi je dat u nastavku:

```
#include "friend.h"
int main()
{
          Friend Joe(260304);
          cout << Joe.getBirthday() << endl;
          Friend John(250406, "Black");
          cout << John.getBirthday() << John.getSurname() << endl;
          return 0;
}
```

# UPOTREBA PREDEFINISANOG (OVERLOADED) KONSTRUKTORA

*Kod kreiranja objekata preklopljeni konstruktori se koriste da inicijalizuju različite skupove promenljivih*

Sledeći primer je modifikovan u odnosu na raniji primer za klasu Macka. Fajl macka2.h:

```
#include "sisar.h"
#include <string>
class Macka : public Sisar {
public:
 Macka(string inicIme)  {
  ime = inicIme;
  cout << "konstruktor1" << endl;
 }
 Macka(string inicIme, int inicStarost )  {
  ime = inicIme;
  starost = inicStarost;
  cout << "konstruktor2"<<endl;
 }
 Macka(string inicIme, int inicStarost, int
inicTezina)  {
  ime = inicIme;
  starost = inicStarost;
  tezina = inicTezina;
  cout << "konstruktor3"<<endl;
 }
 ~Macka(){cout<<"destruktor"<<endl;}
 string getIme() const {return ime;}
 void mjau() const {cout << "mjau" << endl;}
protected:
 string ime;
};
```

Takođe, štampanja su ubačena u konstruktorsku i destruktorsku funkciju, da signaliziraju trenutak u radu programa kada su pozvani. U nastavku je glavni program, napisan u fajlu macka2.cpp, gde se koriste predefinisani konstruktori iz macka2.h fajla:

```
#include "macka2.h"
int main()
{
 Macka MojaMacka("Mackica", 2, 3);
 cout << MojaMacka.getIme();
 cout << MojaMacka.getStarost();
 cout << MojaMacka.getTezina();
 Macka NjegovaMacka("Zika", 2);
 cout << NjegovaMacka.getIme();
 cout << NjegovaMacka.getStarost();
 Macka NjenaMacka("Laki");
 cout << NjenaMacka.getIme();
 NjenaMacka.mjau();
 cout<<endl;
 return 0;
}
```

# Dodavanje i menjanje članova izvedene klase

*izvedena klasa, nadglasavanje, nadglasavajuća i nadglasana f-ja*

- *Uvodna razmatranja*
- *Dodavanje nove funkcionalnosti*
- *Nadglasavanje funkcija (overriding)*
- *Upotreba nadglasavajućih i nadglasanih funkcija*
- *Dodavanje nove funkcionalnosti postojećoj funkciji*

**04**

# UVODNA RAZMATRANJA

*Programeri imaju mogućnost da naslede funkcionalnost bazne klase, dodaju novu funkcionalnost, modifikuju postojeću funkcionalnost ili da sakriju funkcionalnost koja nije potrebna*

U okviru uvodne lekcije o nasleđivanju pomenuli smo da je najveća pogodnost nasleđivanja mogućnost ponovnog korišćenja već napisanog koda. Programeri imaju mogućnost da naslede funkcionalnost bazne klase, dodaju novu funkcionalnost, modifikuju postojeću funkcionalnost ili da sakriju funkcionalnost koja nije potrebna. U nastavku će biti opisano kako se ovo ostvaruje. Krenućemo prvo od proste osnovne klase:

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

class Base
{
protected:
            int m_nValue;

public:
            Base(int nValue)
                        : m_nValue(nValue)
            {
            }

            void Identify() { cout << "I am a Base"
<< endl; }
};
```

Kreirajmo sada izvedenu klasu koja nasleđuje osobine klase Base. S obzirom da želimo da izvedena klasa Derived ima mogućnost da menja vrednost polja m_nValue kada je jedan objekat klase Derived instanciran, mi ćemo kreirati takav konstruktor klase Derived koji će pozivati konstruktor bazne klase kroz listu za inicijalizaciju.

```
class Derived: public Base
{
public:
            Derived(int nValue)
                        :Base(nValue)
            {
            }
};
```

# DODAVANJE NOVE FUNKCIONALNOSTI

*Dodavanje nove funkcionalnosti izvedenoj klasi se vrši jednostavnim deklarisanjem novih funkcija unutar izvedene klase*

S obzirom da imamo pristup kodu bazne klase Base, imamo mogućnost da direktno dodamo novu funkcionalnost klasi Base. Međutim, postojaće slučajevi kada to nećemo hteti ili nećemo moći da uradimo. Najbolji scenario je međutim da izvedemo novu klasu i da novu funkcionalnost pripišemo toj novoj izvedenoj klasi. Dodavanje nove funkcionalnosti izvedenoj klasi se vrši jednostavnim deklarisanjem novih funkcija unutar izvedene klase:

```
class Derived: public Base
{
public:
          Derived(int nValue)
                    :Base(nValue)
          {
          }

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

Korišćenjem funkcije GetValue() pristupamo protected članu bazne klase m_nValue, tako da sada korisnici klase Derived imaju mogućnost da očitaju vrednost njene bazne klase, što prikazuje sledeći segment koda:

```
int main()
{
    Derived cDerived(5);
    cout << "cDerived has value " <<
cDerived.GetValue() << endl;

    return 0;
}
```

Rezultat programa biće:

cDerived has value 5

Iako je očigledno, objekti tipa Base nemaju pristup funkciji GetValue() izvedene klase Derived. Sledeći deo koda neće da radi:

```
int main()
{
    Base cBase(5);
    cout << "cBase has value " << cBase.GetValue()
<< endl;

    return 0;
}
```

Ovo je zato što funkcija GetValue() nije deo klase Base. Funkcija GetValue() pripada samo klasi Derived. S obzirom da je klasa Derived izvedena is Base ona ima pristup funkcijama Base klase, ali suprotno ne važi. Klasa Base nema pristup funkcijama deklarisanim u izvedenoj Derived klasi.

# NADGLASAVANJE FUNKCIJA (OVERRIDING)

*U C++-u imamo mogućnost da u izvedenim klasama kreiramo funkcije sa istim nazivom kao one koje su definisane u baznim klasama ali sa drugačijom funkcionalnošću*

Kada je neka funkcija članica pozvana korišćenjem objekta izvedene klase, kompajler prvo traži da li je ta funkcija deo izvedene klase. Ukoliko nije, on počinje da traži po nivoima baznih klasa (po lancu nasleđivanja) dok ne naiđe na prvu deklaraciju sa istim nazivom. Pogledajmo sledeći primer:

```
int main()
{
    Base cBase(5);
    cBase.Identify();

    Derived cDerived(7);
    cDerived.Identify()

    return 0;
}
```

Rezultat će biti:

```
I am a Base
I am a Base
```

Kada pozovemo funkciju cDerived.Identify(), kompajler proverava da li je funkcija Identify() deklarisana u klasi Derived. Kompajler zatim proverava u baznim klasama (u ovom slučaju klasa Base). Klasa Base ima definisanu funkciju Identify(), pa kompajler upravo nju i koristi.

Međutim, ukoliko smo definisali funkciju Derived::Identify() u izvedenoj Derived klasi, upravo ona će biti i pozvana. Ovo znači da imamo mogućnost da u izvedenim klasama kreiramo funkcije sa istim nazivom kao one koje su definisane u baznim klasama ali sa drugačijom funkcionalnošću!

Radi boljeg razumevanja, biće mnogo preciznije ako definišemo funkciju tako da poziv cDerived.Identify() štampa poruku "I am a Derived". Izmenimo sada funkciju Identify() tako da vraća tačan odziv ako je pozovemo korišćenjem objekta izvedene Derived klase. To je moguće uraditi jednostavnim predefinisanjem funkcije u izvedenoj klasi, tj. definisanjem funkcije sa istim imenom.

```
class Derived: public Base
{
public:
            Derived(int nValue)
                        :Base(nValue)
            {
            }

            int GetValue() { return m_nValue; }

            // Here's our modified function
            void Identify() { cout << "I am a
Derived" << endl; }
};
```

# UPOTREBA NADGLASAVAJUĆIH I NADGLASANIH FUNKCIJA

*Pri nadglasavanju funkcije u izvedenoj klasi, izvedena funkcija ne nasleđuje modifikator pristupa koji je za tu funkciju naveden u baznoj klasi*

Nadglasavajuća funkcija, *overriding method*, se deklariše u izvedenoj klasi, i njena deklaracija mora tačno da se poklapa sa deklaracijom nadglasane metode, *overriden method*, u baznoj klasi. Dakle, obe funkcije, i ona koja nadglasava i ona koja je nadglasana moraju imati isti povratni tip, ime i argumente.

Takođe, ako se nalaze predefinisane funkcije u baznoj klasi, jedna nadglasavajuća funkcija u izvedenoj klasi će nadglasati sve predefinisane funkcije u baznoj klasi, sa istim imenom. Takođe treba napomenuti da pri nadglasavanju funkcije u izvedenoj klasi, izvedena funkcija ne nasleđuje modifikator pristupa koji je za tu funkciju naveden u baznoj klasi. Biće korišćen specifikator pristupa koji je naveden u izvedenoj klasi. Stoga, funkcija koja je definisana kao privatna u baznoj klasi može biti predefinisana kao javna u izvedenoj klasi, i obratno. Primer 'people.h' sa dve preklopljene funkcije u baznoj klasi:

```
class People
{
public:
            void setPhone() { cout << phone <<
endl;}
            void setPhone(int y ) { phone = y;}
protected:
            string surname; int phone;
};
```

Primer 'friend.h' sa nadglasavajućom funkcijom u izvedenoj klasi:

```
#include "people.h"
class Friend : public People
{
public:
            void setPhone(int y)
            {
                        cout << phone << endl;
                        phone = y;
            }
protected:
            int birthday;
};
```

Neka funkcija u baznoj klasi može se pozvati preko eksplicitnog navođenja imena njene klase i imena sa razdvajanjem sa dva puta dve tačke, operatorom :: , i ovo se može upotrebiti kod pozivanja nadglasanih funkcija u baznoj klasi (*overriden by derived class functions*). U nastavku su dati primeri poziva nadglasavajućih i nadglasanih funkcija:

```
Friend John();
John.setPhone(7654321); //poziva potiskujucu finkciju iz
izvedene klase
// use of John.setPhone() –would result in a compiler error
People Joe();
Joe.setPhone(5556666); //poziva potisnutu funkciju iz bazne
klase
//overriden function called
John.People::setPhone();
```

# DODAVANJE NOVE FUNKCIONALNOSTI POSTOJEĆOJ FUNKCIJI

*U C++-u imamo mogućnost da na već postojeću funkcionalnost bazne funkcije dodamo i funkcionalnost koja će biti karakteristična za izvedenu funkciju*

U mnogim slučajevima ne želimo da kompletno zamenimo funkciju bazne klase funkcijom izvedene klase, već da postojećoj funkciji dodamo novu funkcionalnost. Primetimo da u prethodnom primeru funkcija Derived::Identify() potpuno nadglasava (override) funkciju Base::Identify()! U C++-u imamo mogućnost da na već postojeću funkcionalnost bazne funkcije dodamo i funkcionalnost koja će biti karakteristična za izvedenu funkciju. Ovo se ostvaruje tako što definišemo izvedenu funkciju istog naziva kao i bazna funkcija, iz koje zatim pozivamo baznu funkciju tako što navedemo naziv bazne funkcije kojoj prethodi operator rezolucije opsega (::) i naziv bazne klase. U narednom primeru smo predefinisali funkciju Derived::Identify() tako da se iz nje prvo poziva Base::Identify() a zatim se izvršava deo koda koji smo dodatno ugradili.

```
class Derived: public Base {
public:
    Derived(int nValue) :Base(nValue)   { }
    int GetValue() { return m_nValue; }
    void Identify()  {
        Base::Identify(); // call
Base::Identify() first
        cout << "I am a Derived"; // then
identify ourselves
    }
};
```

Ako pozovemo istu main funkciju kao u prethodnim primerima dobićemo sledeći rezultat:

```
I am a Base
I am a Base
I am a Derived
```

Kada se izvršava funkcija cDerived.Identify(), kompajler je vidi kao Derived::Identify(). Međutim, ako pogledamo definiciju, prva stvar koju funkcija Derived::Identify() radi je pozivanje bazne funkcije Base::Identify(), koja štampa poruku "I am a Base". Kada se izvrši funkcija Base::Identify(), Derived::Identify() nastavlja sa izvršavanjem i štampa poruku "I am a Derived". Najbitnija stvar koju treba izvesti iz prethodnog kao zaključak je to da ako želimo da pozovemo funkciju bazne klase koja je predefinisana u izvedenoj klasi, neophodno je da koristimo operator rezolucije opsega (::) kako bi eksplicitno naveli koju verziju funkcije želimo da koristimo. Ukoliko je na primer funkcija Derived::Identify() definisana kao:

```
void Identify()
{
            Identify(); // Note: no scope resolution!
            cout << "I am a Derived"; // then identify
ourselves
}
```

ućiće se u beskonačnu petlju jer je funkcija Identify() ustvari Derived::Identify() pa će funkcija zvati samu sebe!

# Sakrivanje članova u izvedenoj klasi

*nasleđivanje, izvedene klase, sakrivanje članova*

- *Sakrivanje metoda bazne klase*
- *Sakrivanje podataka bazne klase*

05

# SAKRIVANJE METODA BAZNE KLASE

*U C++-u je moguće predefinisati javnu funkciju bazne klase tako da ona postane privatna u izvedenoj klasi, pa joj korisnici klase neće moći pristupiti*

U C++-u nije moguće ukloniti funkcionalnost klase, ali je moguće sakriti je. Kao što je prethodno navedeno, ukoliko predefinišemo funkciju, ona će koristiti specifikator pristupa koji je deklarisan u izvedenoj klasi. Stoga je moguće predefinisati javnu funkciju bazne klase tako da ona postane privatna u izvedenoj klasi, pa joj korisnici klase neće moći pristupiti. Međutim, C++ vam takođe daje mogućnost da izmenite specifikator pristupa članu bazne klase u izvedenoj klasi bez predefinisanja metode. Ovo se ostvaruje jednostavnim navođenjem člana klase (korišćenjem operatora razrešavanja opsega ::) da bi se izmene zapamtile u izvedenoj klasi. Posmatrajmo na primer, sledeći oblik klase Base:

```
class Base
{
private:
            int m_nValue;

public:
            Base(int nValue)
                        : m_nValue(nValue)
            {
            }

protected:
            void PrintValue() { cout << m_nValue; }
};
```

S obzirom da je funkcija Base::PrintValue() deklarisana kao protected, ona može biti pozvana iz metoda klase Base ili iz metoda izvedene klase. Javnost nema pristup ovoj metodi. Definišimo sada klasu Derived u kojoj je specifikator pristupa metodi PrintValue() promenljen u public:

```
class Derived: public Base
{
public:
    Derived(int nValue)
        : Base(nValue)  { }
    Base::PrintValue;
};
```

Ovo znači da će sledeći program raditi bez grešaka:

```
int main()
{
            Derived cDerived(7);

            // PrintValue is public in Derived, so
this is okay
            cDerived.PrintValue(); // prints 7
            return 0;
}
```

# SAKRIVANJE PODATAKA BAZNE KLASE

*Ova metodologija dozvoljava da uzmemo loše kreiranu baznu klasu (po kriterijumima OOP) i da zatim njene podatke enkapsuliramo u izvedenoj klasi*

Ono što možemo primetiti kod prethodnog primera je to da u definiciji funkcije Base::PrintValue nije naveden funkcijski operator (). Istu metodologiju možemo koristiti da bi javni podaci klase postali privatni:

```
class Base
{
public:
            int m_nValue;
};
class Derived: public Base
{
private:
            Base::m_nValue;
public:
            Derived(int nValue)
            {
                        m_nValue = nValue;
            }
};
int main()
{
            Derived cDerived(7);

            // The following won't work because
m_nValue has been redefined as private
            cout << cDerived.m_nValue;

            return 0;
}
```

Primetimo da nam prethodna metodologija dozvoljava da uzmemo loše kreiranu baznu klasu (po kriterijumima OOP) i da zatim njene podatke enkapsuliramo u izvedenoj klasi.

Napomena: moguće je promeniti specifikatore pristupa onim članovima bazne klase koji su vidljivi izvedenoj klasi. Stoga, nije moguće promeniti specifikator pristupa članu bazne klase iz private u protected ili public, jer izvedena klasa nema pristup privatnim članovima bazne klase.

# Višestruko nasleđivanje

*nasleđivanje, višestruko nasleđivanje, problem dijamanta*

- *Uvodna razmatranja*
- *Uvodna razmatranja*
- *Problemi kod višestrukog nasleđivanja*
- *Problem dijamanta kod višestrukog nasleđivanja*

**06**

*Višestruko nasleđivanje omogućava da izvedena klasa nasledi članove više od jednog roditelja*

Svi primeri nasleđivanja koje smo predstavili do sada su bili primeri jednostrukog nasleđivanja, što znači da smo imali klase koje su nasledile osobine i ponašanja od samo jednog roditelja. Višestruko nasleđivanje omogućava da izvedena klasa nasledi članove više od jednog roditelja.

Pretpostavimo da hoćemo da napišemo program koji će služiti za praćenje inormacija o statusu grupe učitelja. Učitelj je naravno osoba, ali je istovremeno i radnik. U ovom primeru je moguće koristiti principe višestrukog nasleđivanja kako bi kreirali klasu Teacher koja istovremeno nasleđuje osobine klasa Person i Employee. Da bi se koristilo višestruko nalseđivanje jednostavno pri deklaraciji klase treba navesti sve bazne klase razdvojene zarezima. Sintaksa ima sledeći oblik:

```
class derived-class: access baseA, access baseB....
```

gde se access odnosi na specifikator public, protected ili private i biće priključen svakoj baznoj klasi iz koje izvodimo nasleđenu klasu. Ukoliko na primer u programu imamo specijalnu klasu Output koja vrši štampu na ekran i želimo da naše klase Rectangle i Triangle naslede osobine klase Output kao dodatak na osobine klase Polygon, možemo napisati:

```
class Rectangle: public Polygon, public Output;
class Triangle: public Polygon, public Output;
```

U nastavku je dat kompletan primer:

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

class Polygon {
protected:
 int width, height;
public:
 Polygon (int a, int b) : width(a), height(b)
{}
};

class Output {
public:
 static void print (int i);
};

void Output::print (int i) {
 cout << i << '\n';
}

class Rectangle: public Polygon, public Output
{
public:
 Rectangle (int a, int b) : Polygon(a,b) {}
 int area ()
 { return width*height; }
};
```

# UVODNA RAZMATRANJA

*Višestruko nasleđivanje omogućava da izvedena klasa nasledi članove više od jednog roditelja*

Svi primeri nasleđivanja koje smo predstavili do sada su bili primeri jednostrukog nasleđivanja, što znači da smo imali klase koje su nasledile osobine i ponašanja od samo jednog roditelja. Višestruko nasleđivanje omogućava da izvedena klasa nasledi članove više od jednog roditelja.

Pretpostavimo da hoćemo da napišemo program koji će služiti za praćenje inormacija o statusu grupe učitelja. Učitelj je naravno osoba, ali je istovremeno i radnik. U ovom primeru je moguće koristiti principe višestrukog nasleđivanja kako bi kreirali klasu Teacher koja istovremeno nasleđuje osobine klasa Person i Employee. Da bi se koristilo višestruko nalseđivanje jednostavno pri deklaraciji klase treba navesti sve bazne klase razdvojene zarezima. Sintaksa ima sledeći oblik:

```
class derived-class: access baseA, access baseB....
```

gde se access odnosi na specifikator public, protected ili private i biće priključen svakoj baznoj klasi iz koje izvodimo nasleđenu klasu. Ukoliko na primer u programu imamo specijalnu klasu Output koja vrši štampu na ekran i želimo da naše klase Rectangle i Triangle naslede osobine klase Output kao dodatak na osobine klase Polygon, možemo napisati:

```
class Rectangle: public Polygon, public Output;
class Triangle: public Polygon, public Output;
```

U nastavku je dat kompletan primer:

```
class Triangle: public Polygon, public Output {
public:
 Triangle (int a, int b) : Polygon(a,b) {}
 int area ()
 { return width*height/2; }
};

int main () {
 Rectangle rect (4,5);
 Triangle trgl (4,5);
 rect.print (rect.area());
 Triangle::print (trgl.area());
 return 0;
}
```

# PROBLEMI KOD VIŠESTRUKOG NASLEĐIVANJA

*Jedan od problema višestrukog nasleđivanja je dvosmislenost, koja se javlja u slučaju kada bazne klase sadrže funkcije sa istim nazivima*

Iako višestruko nasleđivanje na prvi pogled deluje kao jednostavno proširenje jednostrukog nasleđivanja, ono uvodi puno novih problema koje mogu značajno da uvećaju kompleksnost programa i od samog postupka nasleđivanja naprave noćnu moru. U nastavku su razmotrene neko od pomenutih situacija. Kao prvo, može se javiti dvosmislenost, i to u slučaju kada bazne klase sadrže funkcije sa istim nazivima. Definišimo na primer klase USBDevice i NetworkDevice:

```
class USBDevice
{
private:
 long m_lID;
public:
 USBDevice(long lID)
  : m_lID(lID) { }
 long GetID() { return m_lID; }
};

class NetworkDevice
{
private:
 long m_lID;
public:
 NetworkDevice(long lID)
  : m_lID(lID) { }
 long GetID() { return m_lID; }
};
```

Definišimo sada klasu koja nasleđuje obe kreirane klase:

```
class WirelessAdaptor: public USBDevice, public
NetworkDevice
{
public:
            WirelessAdaptor(long lUSBID, long lNetworkID)
                        : USBDevice(lUSBID),
NetworkDevice(lNetworkID)
            {
            }
};
```

Pretpostavimo da u glavnom programu imamo sledeće naredbe:

```
WirelessAdaptor c54G(5442, 181742);
cout << c54G.GetID(); // Which GetID() do we call?
```

Kada se izvršava funkcija c54G.GetID(), kompajler provarava da li je ona definisana u klasi WirelessAdaptor, ali je ne pronalazi. Kompajler zatim traži da li neka od baznih klasa sadrži ovu funkciju, i javlja se problem. Problem je taj jer objekat c54G ustvari sadrži dve funkcije GetID(): jedna je nasleđena iz klase USBDevice, a druga je nasleđena iz klase WirelessDevice. Stoga je ovaj poziv funkcije dvosmislem pa će kompajler prijaviti grešku. Međutim, postoji način da se zaobiđe ovaj problem: moguće je eksplicitno navesti koju funkciju želimo da pozovemo, i to radimo na sledeći način:

```
WirelessAdaptor c54G(5442, 181742);
cout << c54G.USBDevice::GetID();
```

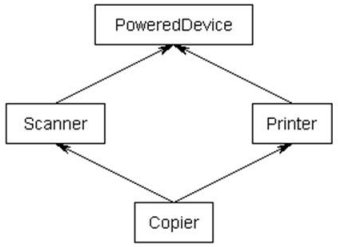
# PROBLEM DIJAMANTA KOD VIŠESTRUKOG NASLEĐIVANJA

*Problem „dijamanta“ nastaje kada neka klasa nasleđuje osobine dve bazne klase, a te bazne klase su izvedene iz iste osnovne klase*

Iako je u prethodnom primeru upotrebljen prost način za razrešavanje problema, dvosmislenost je i dalje tu a naročito dolazi do izražaja ako naša klasa nasleđuje osobine od 4 ili više osnovnih klasa. U takvim situacijama raste verovatnoća da se jave konflikti u nazivima, koji se moraju pojedinačno rešavati.

Drugi, i mnogo ozbiljniji je problem „dijamanta“ koji nastaje kada neka klasa nasleđuje osobine dve bazne klase, a te bazne klase su izvedene iz iste osnovne klase. Ovo vodi do pojave takozvanog dijamantskog oblika nasleđivanja. Uzmimo u obzir sledeći set klasa:

```
class PoweredDevice
{
};

class Scanner: public PoweredDevice
{
};

class Printer: public PoweredDevice
{
};

class Copier: public Scanner, public Printer
{
};
```




Slika-1 Problem dijamanta kod višestrukog nasleđivanja

Klase Scanners i Printers su obe izvedene iz PoweredDevice. Međutim, mašina za kopiranje Copier može da ima funkcionalnost i skenera i štampača, pa istovremeno nasleđuje osobine klasa Scanners i Printers (Slika-4).
Kao što znamo, većina problema koji se mogu rešiti upotrebom višestrukog nasleđivanja može se jednako dobro rešiti pomoću jednostrukog nasleđivanja. Mnogi objektno orijentisani jezici ni ne podržavaju višestruko nasleđivanje. Mnogi relativno noviji jezici kao što su Java i C # ograničavaju klase na jednostrukom nasleđivanju, ali dozvoljavaju višestruko nasleđivanje interfejsa. Glavna ideja onemogućavanja višestrukog nasledstva u ovim jezicima je činjenica da ono jednostavno čini jezike previše složenim, a na kraju izaziva nove problema umesto da rešave postojeće. Mnogi autori i iskusni programeri veruju da višestruko nasleđivanje treba izbegavati po svaku cenu zbog mnogih potencijalnih problema koje ono donosi. Međutim postoje situacije kada je višestruko nasleđivanje najbolje rešenje, ali u tim slučajevima ga treba koristiti ekstremno promišljeno.

# Kompozicija

*kompozicija, sastav, objekat kao član klase, slaganje klasa*

- *Uvod u kompoziciju*
- *Upotreba klasa u drugim klasama – Korišćenje liste za inicijalizaciju*
- *Kompletan primer kompozicije – Klasa Point2D*
- *Kompletan primer kompozicije – Klasa Creature*
- *Kompletan primer kompozicije – Glavni program*

**07**

# UVOD U KOMPOZICIJU

*Proces izgradnje složenog objekta od jednostavnijih objekata se naziva kompozicija (object composition)*

U stvarnom životu, složeni objekti se često grade od manjih, jednostavnijih objekata. Na primer , automobil je napravljen pomoću metalne kontrukcije, motora, guma, prenosa, volana i ostalih brojnih delova. Personalni računar je izgrađen od procesora, matične ploče, memorije, itd ... Čak su i ljudi izgrađeni od manjih delova: imate glavu, telo, noge, ruke i tako dalje. Ovaj proces izgradnje složenog objekta od jednostavnijih objekata se naziva kompozicija (object composition).

Do sada, sve klase koje smo koristili u našim primerima su imale članice koje su bile primitivnih ili ugrađenih tipova podataka (npr . int, double ). Iako je ovo generalno dovoljno za projektovanje i sprovođenje male, jednostavne klase, ubrzo postaje otežavajuće za konstrukciju složenijih klasa, posebno onih građenih od mnogih pod-delova. Da bi se olakšala izgradnja složenih klase iz jednostavnijih, C ++ nam omogućava da obavljamo kompoziciju objekata na vrlo jednostavan način - pomoću klase kao promenljive članice u drugim klasama. Kompozicija klasa stoga označava upotrebu jedne ili više klasa u okviru definicije tj. deklaracije druge klase. Kada je neki član neke nove klase u stvari objekt neke druge klase, kaže se: nova klasa je složena klasa, '*composite class*'. Dakle, kompozitna (složena) klasa X sadrži članove koji su objekti klase Y:

```
class X
{
            ...
public/private/protected:
           Y yy;
            ...
};
```

Pogledamo još jedan primer kako se pravi kompozicija. Ako želimo da kreiramo klasu  za personalni računar, mogli bismo to uraditi na sledeći način ( pod pretpostavkom da smo već kreirali klase za CPU, matičnu ploču, i RAM):

```
#include "CPU.h"
#include "Motherboard.h"
#include "RAM.h"

class PersonalComputer
{
private:
           CPU m_cCPU;
           Motherboard m_cMotherboard;
           RAM m_cRAM;
};
```

# UPOTREBA KLASA U DRUGIM KLASAMA – KORIŠĆENJE LISTE ZA INICIJALIZACIJU

*Veoma je čest slučaj kod kompozicije da se članovi klase, koji su objekti druge klase, umesto korišćenjem podrazumevajućeg kontruktora inicijalizuju korišćenjem liste za inicijalizaciju*

Kao i u programskom jeziku Java, objekat jedne klase može da bude član objekta druge klase. U nastavku će biti opisan postupak poziva konstuktora kada se desi upravo takav slučaj.

Za podatke članove klase koji su osnovnih primitivnih tipova, ne postoji bitna razlika u tome na koji način se vrši definicija konstruktora, s obzirom da oni nisu podrazumevano inicijalizovani, ali za članove klase koji su objekti (objekti čiji je tip neka druga klasa), oni moraju biti definisani korišćenjem podrazumevanog konstruktora.

Inicijalizacija svih članova klase korišćenjem podrazumevajućeg konstruktora u većini slučaja nije baš najsrećnije rešenje: u nekim slučajevima, ovo je samo gubljenje vremena (kada se član zatim reinicijalizuje u nekom drugom konstruktoru), ali u nekim drugim situacijama, kreiranje korišćenjem podrazumevajućeg konstruktora nije čak ni moguće (kada klasa ne sadrži podrazumevajući konstruktor). U ovim slučajevima, podaci članovi treba biti inicijalizovani korišćenjem liste za inicijalizaciju. U ovom primeru, klasa Cylinder ima član koji je objekat druge klase (promenljiva base je tipa Circle):

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

class Circle {
    double radius;
  public:
    Circle(double r) : radius(r) { }
    double area() {return radius*radius*3.14159265;}
};

class Cylinder {
    Circle base;
    double height;
  public:
    Cylinder(double r, double h) : base (r),
height(h) {}
    double volume() {return base.area() * height;}
};

int main () {
  Cylinder foo (10,20);

  cout << "foo's volume: " << foo.volume() << '\n';
  return 0;
}
```

S obzirom da objekti klase Circle mogu biti konstruisani samo pomoću konstruktora sa parametrima,konstruktor klase Cylinder mora da pozove konstruktor objekta base, a jedini način da se ovo uradi je korišćenjem liste za inicijalizaciju članova.

# KOMPLETAN PRIMER KOMPOZICIJE – KLASA POINT2D

*Mnoge igre i simulacije imaju tvorevine i objekte koji mogu da se pomeraju na tabli, mapi ili ekranu. Kreirana klasa Point2D služi kao lokacija postojećeg objekta*

Dok su prethodni primer korisni u davanju opšte slike kako funkcioniše kompozicija, hajde da uradimo kompletan primer u cilju detaljnijeg uvida. Mnoge igre i simulacije imaju tvorevine (creature) i objekte koji mogu da se pomeraju na tabli, mapi ili ekranu.  U narednom primeru ćemo kreirati neku klasu koja će služiti kao lokacija postojećeg objekta.

Kreirajmo prvo klasu za tačku, Point. Naša tačka će živeti u 2D prostoru tako da će naša tačka imati dve dimenzije, X i Y. Pretpostavićemo da se prostor sastoji iz celih brojeva, tako da ćemo koristiti tip int. Deklaraciju klase ćemo smestiti u fajlu Point2D.h:

```
#ifndef POINT2D_H
#define POINT2D_H
#include <iostream>
class Point2D
{
private:
 int m_nX;
 int m_nY;
public:
 Point2D()
  : m_nX(0), m_nY(0)  { }
 Point2D(int nX, int nY)
  : m_nX(nX), m_nY(nY) { }
 friend std::ostream& operator<<(std::ostream&
out, const Point2D &cPoint)
 {
  out << "(" << cPoint.GetX() << ", " <<
cPoint.GetY() << ")";
  return out;
 }
 void SetPoint(int nX, int nY)
 {
  m_nX = nX;
  m_nY = nY;
 }
 int GetX() const { return m_nX; }
 int GetY() const { return m_nY; }
};
#endif
```

# KOMPLETAN PRIMER KOMPOZICIJE – KLASA CREATURE

*Kreirana klasa Creature može da posluži kao tvorevina (creature) neke igrice. Korišćenjem koncepta kompozije objekat klase Point2D postaje podatak član klase Creature*

Možemo primetiti da zbog jednostavosti primera, sve funkcije implementiramo u okviru definicije klase tako da ne kreiramo fajl Point2D.cpp. Kreirajmo sada odgovarajuću klasu koja će da služi za tvorevine igrice, i nazovimo je Creature. Klasa Creature će imati nekoliko osobina: ime (string) i lokaciju koja će biti određena klasom Point2D. Definiciju klase Creature ćemo napisati u fajlu Creature.h:

```
#ifndef CREATURE_H
#define CREATURE_H

#include <iostream>
#include <string>
#include "Point2D.h"

class Creature {
private:
 std::string m_strName;
 Point2D m_cLocation;

 Creature() { }
public:
 Creature(std::string strName, const Point2D
&cLocation)
  : m_strName(strName), m_cLocation(cLocation){
}
 friend std::ostream& operator<<(std::ostream&
out, const Creature &cCreature) {
  out << cCreature.m_strName.c_str() << " is at
" << cCreature.m_cLocation;
  return out;
 }
 void MoveTo(int nX, int nY) {
  m_cLocation.SetPoint(nX, nY);
 }
};
```

# KOMPLETAN PRIMER KOMPOZICIJE – GLAVNI PROGRAM

*Funkcionalnost kreiranih klasa i samog procesa kompozicije testiramo korišćenjem odgovarajućih naredbi u glavnom programu*

Konačno imamo fajl main.cpp:

```
#include <string>
#include <iostream>
#include "Creature.h"
int main()
{
 using namespace std;
 cout << "Enter a name for your creature: ";
 std::string cName;
 cin >> cName;
 Creature cCreature(cName, Point2D(4, 7));
 while (1)  {
  cout << cCreature << endl;
  cout << "Enter new X location for creature (-
1 to quit): ";
  int nX=0;
  cin >> nX;
  if (nX == -1)
   break;
  cout << "Enter new Y location for creature (-
1 to quit): ";
  int nY=0;
  cin >> nY;
  if (nY == -1)
   break;
  cCreature.MoveTo(nX, nY);
 }
 return 0;
}
```

Nakon kompajliranja i pokretanja programa dobićemo sledeći rezultat:

```
Enter a name for your creature: Marvin
Marvin is at (4, 7)
Enter new X location for creature (-1 to quit): 6
Enter new Y location for creature (-1 to quit): 12
Marvin is at (6, 12)
Enter new X location for creature (-1 to quit): 3
Enter new Y location for creature (-1 to quit): 2
Marvin is at (3, 2)
Enter new X location for creature (-1 to quit): -1
```

# ZAŠTO KORISTITI KOMPOZICIJU?

*Posao složene klase nije da zna intimne detalje o klasama koje je sačinjavaju, već da brine o tome kako da koordinira protok podataka i obezbedi da svaka od podklasa zna šta treba da radi*

Umesto da koristimo klasu Point2D kako bi implementirali lokaciju klase Creature, mogli smo jednostavno dodati dva nova podatka tipa int u okviru definicije klase Creature kao i napisan kod koji bi omogućio menjanje i očitavanje pozicije. Međutim, korišćenje kompozicije ima brojne pogodnosti:

- Svaka pojedinačna klasa može ostati relativno jednostavna i jasna, fokusirana na obavljanje jednog specifičnog zadatka. To omogućava jednostavnije pisanje i lakše razumevanje klasa.
- Svaki podobjekat može biti samostalan, što omogućava ponovno korišćenje gotovog objekta u drugim klasama. Na primer, mogli bismo ponovo koristiti našu Point2D klasu u potpuno drugačiju svrhu. Ili, ako je u našoj klasi Creature ikada potreban još jedan podatak (na primer, destinacija do koje pokušavamo da dođemo), možemo jednostavno dodati još jednu Point2D promenljivu članicu.

Jedno od pitanja koje početnici u programiranju često postavljaju je „Kada da koristim kompoziciju umesto da kreiram novi klasu“. Ne postoji 100% odgovor na ovo pitanje. Međutim, dobro pravilo je da svaka klasa treba da se gradi ca ciljem bi se ostvario jedan specifičan zadatak.

U slučaju našeg primera, klasa Creature ne bi trebalo da brine o tome kako je implementirana klasa Point, ili na koji način se čuva podatak o nazivu. Posao klase Creature nije da zna te intimne detalje, već da brine o tome kako da koordinira protok podataka i obezbedi da svaka od podklasa zna šta treba da radi. Na pojedinačnim podklasama je da brinu o tome kako će one uraditi njihov deo posla.

# Agregacija

*agregacija, skup, klasa u klasi, pokazivači*

- *Uvod u agregaciju*
- *Primer agregacije*
- *Razlike između kompozicije i agregacije*

08

# UVOD U AGREGACIJU

*Agregacija je specifična vrsta kompozicije u kojoj se ne podrazumeva vlasništvo kompleksnog objekta nad njegovim podobjektima*

U prethodnoj sekciji o kompoziciji, rekli smo da su kompozicije složene klase koje sadrže druge podklase kao promenljive članice. Pored toga, u kompoziciji kompleksan objekat "poseduje" sve podobjekte od kojih je sastavljen. Kada se kompozicija uništava, uništavaju se i svi podobjekti. Na primer, ako uništimo auto, onda će njegova konstrukcija, motor, i drugi delovi biti uništeni.

Agregacija je specifična vrsta kompozicije u kojoj se ne podrazumeva vlasništvo kompleksnog objekta nad njegovim podobjektima. Kada je uništena agregacija, njeni podobjekti neće biti uništeni.

Na primer, razmotrimo katedru za matematiku u okviru neke škole. S obzirom da katedra ne poseduje profesore (oni samo rade tamo), odeljenje treba da bude agregacija. Kada se uništi objekat koji je tipa Odeljenje, nastavnici treba i dalje da postoje nezavisno (oni mogu dobiti zaposlenje u drugim katedrama).

S obzirom da su agregacije samo poseban tip kompozicije, one se implementiraju skoro identično, a razlika između njih je uglavnom semantička. Kod kompozicija, mi obično dodajemo naše podklase u sastav korišćenjem bilo normalne promenljive ili pokazivača gde se proces alokacije i dealokacije vrši u samoj složenoj klasi (kompoziciji).

U agregaciji (skupu), mi takođe dodajemo i druge podklase našoj kompleksnoj agregatnoj klasi kao promenljive članice. Međutim, ove promenljive članice su obično ili reference ili pokazivači koji se koriste da se pokaže na objekte koji su kreirani van okvira klase. Shodno tome, agregatna klasa obično ili prima objekte na koje će pokazivati preko parametara konstruktora, ili je prazna pri kreiranju a naknadno se dodaju podobjekti preko pristupnih funkcija ili operatora.

Pošto ovi objekti podklasa žive van opsega klase, kada je uništena klasa, pokazivačka ili referencna promenljiva članica će biti uništena ali objekti podklase će i dalje postojati.

# PRIMER AGREGACIJE

*S obzirom da su agregacije samo poseban tip kompozicije, one se implementiraju skoro identično, a razlika između njih je uglavnom semantička*

Pogledajmo sada detaljnije klase Teacher and Department u cilju boljeg razumevanja kompozicije i agregacije:

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

class Teacher
{
private:
            string m_strName;
public:
            Teacher(string strName)
                        : m_strName(strName)
            {
            }

            string GetName() { return m_strName; }
};

class Department
{
private:
            Teacher *m_pcTeacher; // This dept holds
only one teacher

public:
            Department(Teacher *pcTeacher=NULL)
                        : m_pcTeacher(pcTeacher)
            {
            }
};
```

Glavni program u kome testiramo klase će biti napisan kao:

```
void main()
{
            Teacher *pTeacher = new Teacher("Bob");
            {
                        Department cDept(pTeacher);
            }

            delete pTeacher;
}
```

U ovom slučaju objekat pTeacher je kreiran nezavisno od cDept, a zatim je prosleđen konstruktoru objekta cDept. Imajte na umu da klasa Department koristi listu za inicijalizaciju kako bi postavili vrednosti članu m_pcTeacher na pTeacher, koju smo prosledili kao argument konstruktora. Kada je uništena promenljiva cDept, uništen je i pokazivač m_pcTeacher. Međutim, pTeacher nije dealociran, tako da još uvek postoji sve dok se nezavisno ne uništi.

# RAZLIKE IZMEĐU KOMPOZICIJE I AGREGACIJE

*Kompozicija je, za razliku od agregacije, odgovorna za kreiranje / uništavanje njenih podklasa.*
*Agragacija uglavnom koristi pokazivače koji pokazuju na objekte podklase*

Sumirajmo sada razlike između kompozicije i agregacije.

Kompozicija:

- Obično koristi normalne promenljive članice
- Može koristiti pokazivače ukoliko se radi sa dinamičkim alociranjem / dealociranjem memorije.
- Odgovorna je za kreiranje / uništavanje podklasa

Agregacija:

- Obično koristi pokazivačke promenljive članice koje pokazuju na objekte podklase, pri čemu objekti klase žive van opsega agregatne klase.
- Može koristiti reference da pokazuje na objekte koji žive van opsega agregatne klase
- Nije odgovorna za kreiranje / uništavanje podklasa

# Vežbe

*pokazivači, nasleđivanje, kompozicija, agregacija*

- *Zadaci za samostalan rad*

**09**

# NASLEĐIVANJE – PRIMER1

*Posmatrajmo sledeći primer gde imamo baznu klasu Shape i izvedenu klasu Rectangle. Cilj primera je prikaz nasleđivanja u programiranju*

Posmatrajmo sledeći primer gde imamo baznu klasu Shape i izvedenu klasu Rectangle

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

class Shape {
   public:
      void setWidth(int w)      {
         width = w;
      }
      void setHeight(int h)      {
         height = h;
      }
   protected:
      int width;
      int height;
};

class Rectangle: public Shape {
   public:
      int getArea()      {
         return (width * height);
      }
};
```

Glavni program možemo napisati kao:

```
int main(void)
{
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

Nakon kompajliranja i izvršavanja programa dobijamo sledeći rezultat:

```
Total area: 35
```

# NASLEĐIVANJE – PRIMER2

*U nastavku je dat primer nasleđivanja gde imamo baznu klasu Fakultet i izvedenu klasu VišaŠkola. Cilj primera je prikaz nasleđivanja u programiranju*

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

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

class Visaskola : public Fakultet{
public:
            int bivsi_studenti;
            int infoz(void);
            void inicijalizuj(void);
private:
            int poziv;
};
```

Za klasu Fakultet definišemo funkcije info i infoz:

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

Za klasu Visaskola definisemo nadglasavajuću funkciju infoz i novu funkciju inicijalizuj:

```
int Visaskola::infoz(void)
{
            poziv++;
            return poziv;
}
void Visaskola::inicijalizuj(void)
{
            //    zarada=0.;        nije pristupno,
bilo je privatno za Fakultet !
            poziv=0;
            bivsi_studenti=0.;
            br_zaposlenih=0;
            strcpy(adresa,"Adresa nije poznata");
```

Glavni program je:

```
void main(void)
{
            Visaskola tehnicka;
            tehnicka.inicijalizuj();
            tehnicka.br_zaposlenih=50;
            tehnicka.info();
            printf("poziv broj:
%d\n",tehnicka.infoz());
            printf("poziv broj:
%d\n",tehnicka.infoz());
```

Funkcija infoz je preklopljena u izvedenoj klasi i sada ona ima zadatak da prebrojava broj poziva funkcije infoz u objektu izvedene klase Visaskola.

# VIŠESTRUKO NASLEĐIVANJE – PRIMER

*U nastavku je dat primer gde imamo klase za trougao i pravougaonik koje nasleđuju osobine klase Cpoligon i dodatne klase Cprint koja omogućava štampu podataka*

Definicija osnovnih klasa:

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

class Cpolygon
{
protected:
          int width, height;
public:
          void input_values (int one, int two)
          {
                    width=one;
                    height=two;
          }
};

class Cprint
{
public:
          void printing (int output);
};

void Cprint::printing (int output)
{
          cout << output << endl;
}
```

Definicija izvedenih klasa, i glavni main program:

```
class Crectangle: public Cpolygon, public Cprint
{
public:
          int area ()
          {
                    return (width * height);
          }
};

class Ctriangle: public Cpolygon, public Cprint
{
public:
          int area ()
          {
                    return (width * height / 2);
          }
};

int main ()
{
          Crectangle rectangle;
          Ctriangle triangle;
          rectangle.input_values (2,2);
          triangle.input_values (2,2);
          rectangle.printing (rectangle.area());
          triangle.printing (triangle.area());
          return 0;
}
```

# Zadaci za samostalan rad

*pokazivači, nasleđivanje, nadglasavanje, kompozicija*

- *Zadaci za samostalno vežbanje*

09

# ZADACI ZA SAMOSTALNO VEŽBANJE

*Na osnovu materijala za ovu nedelju uraditi samostalno sledeće zadatke:*

1. Napraviti dve klase. Prva klasa je Vozac koja sadrži ime i prezime i broj godina vozača, a druga klasa je Vozilo koja sadrži naziv, registarski broj vozila i vozača (objekat klase Vozac). Prikazati rad klasa korišćenjem pokazivača na klase.

2. Za potrebu kreiranja C++ aplikacije koja podržava rad jednog lanca supermaketa neophodno je kreirati sledeće klase:

• Supermarket - koja od podataka ima naziv, adresu, broj telefona i matični broj.

• Artikal – koja od podataka ima naziv, cenu, šifru proizvoda i naziv proizvođača.

• Prehrambeni proizvod; Svaki prehrambeni prozvod je artikal. Za svaki prehrambeni proizvod čuvamo i rok trajanja i energetsku vrednost.

• Dnevna štampa; Dnevna štampa je u ponudi kao i ostali artikli. Za nju čuvamo datum izdavanja.

3. Omogućiti korisniku da kreira niz od 5 članova koji su tipa Supermarket (iz zadatka broj 2) sa osnovnim podacima (naziv, adresa, broj telefona i matični broj). Nakon unosa elemenata niza, potrebno je sačuvati unete podatke u tekstualni fajl na disk. Podatke o supermarketima upisivati u tekstualni fajl sa ekstenzijom *.txt. Potrebno je da se jednan supermarket nalazi u jednom redu u .txt datoteci, a sve vrednosti su međusobno razdvojene zarezom. Primer:

• *Supermarket na čošku, Nemanjina 2,011/6643-170,123456*

4. Napraviti klasu Životinja koja od atributa ima: ime životinje i rasu. Napraviti metodu zvuk koja treba da ispisuje zvuk koji životinja ispušta. Napraviti klasu Pas i klasu Zmija koje nasleđuju klasu Životinja a potom promeniti zvuk u zavisnosti od životinje.

5. Napraviti klase na osnovu sledećeg teksta. Klijenti u našoj banci imaju Račun. Svaki račun ima svoje stanje. Na račun se mogu uplatiti pare a takođe pare se mogu i podići sa njega. Imamo tri tipa računa: Devizni račun, Dinarski račun kao i Omladinski Račun. Pri uplati na omladinski račun korisnik gubi 2% uplaćenog. Pri uplati na dinarski račun korisnik gubi 5% od uplaćenog dok kod deviznog gubi 5% i pri skidanju i pri uplati

# Zaključak

*11*

# REZIME

*Na osnovu svega obrađenog možemo zaključiti sledeće:*

Objektima neke klase je moguće pristupiti korišćenjem pokazivača. Da bi se pristupilo članu klase korišćenjem pokazivača na klasu, koristi se operator pristupa **->**. Kao i kod ostalih tipova podataka, operatore new i delete možemo koristiti za dinamičko alociranje memorije neophodne za čuvanje objekta, odnosno oslobađanje, respektivno.

Nasleđivanje omogućava kreiranje vrlo složenih klasa krećući se od osnovih klasa, što olakšava kreiranje i održavanje aplikacije. Programer ima mogućnost da navede da će nova klasa naslediti osobine postojeće klase. Postojeća klasa se naziva osnovna ili bazna klasa dok se nove klase nazivaju izvedene klase. Kada jedna klasa nasleđuje drugu klasu, članovi izvedene klase mogu da pristupe zaštićenim članovima osnovne klase, ali nemaju pristup privatnim članovima. Stoga, članove bazne klase treba definisati kao privatne ukoliko želimo da sprečimo pristup od strane funkcija izvedene klase. Izvedena klasa automatski nasleđuje javne i zaštićene članove bazne klase. Izvedene klase kombinuju nasleđene i izvedene elemente tj. članove.

Programeri imaju mogućnost da naslede funkcionalnost bazne klase, dodaju novu funkcionalnost, modifikuju postojeću funkcionalnost ili da sakriju funkcionalnost koja nije potrebna. Dodavanje nove funkcionalnosti izvedenoj klasi se vrši jednostavnim deklarisanjem novih funkcija unutar izvedene klase. U C++-u imamo mogućnost da u izvedenim klasama kreiramo funkcije sa istim nazivom kao one koje su definisane u baznim klasama ali sa drugačijom funkcionalnošću. Ovaj postupak se naziva nadglasavanje (overriding) funkcija. U C++-u je moguće predefinisati javnu funkciju bazne klase tako da ona postane privatna u izvedenoj klasi, pa joj korisnici klase neće moći pristupiti.

Višestruko nasleđivanje omogućava da izvedena klasa nasledi članove više od jednog roditelja. Da bi se koristilo višestruko nasleđivanje jednostavno pri deklaraciji klase treba navesti sve bazne klase razdvojene zarezima.

Proces izgradnje složenog objekta od jednostavnijih objekata se naziva kompozicija (object composition). Veoma je čest slučaj kod kompozicije da se članovi klase, koji su objekti druge klase, umesto korišćenjem podrazumevajućeg kontruktora inicijalizuju korišćenjem liste za inicijalizaciju. Agregacija je specifična vrsta kompozicije u kojoj se ne podrazumeva vlasništvo kompleksnog objekta nad njegovim podobjektima.