15. Objektno orijentirano programiranje

Objektno temeljeno programiranje je metoda programiranja kojoj je temeljni princip da se klasa definira kao samostalna programska cjelina. Pri tome je poželjno koristiti

- **princip enkapsulacije – združivanje funkcija i podataka s definiranim javnim sučeljem i implementacijom koja je skrivena od korisnika,**
- **kompoziciju objekata – članovi klase mogu se deklarirati pomoću postojećih klasa**
- **generičko definiranje klasa**

Ove tehnike smo do sada koristili.

Pod objektno orijentiranim programiranjem (OOP) podrazumijeva se metoda programiranja kojom se definiranje neke klase vrši korištenjem svojstava postojećih klasa. Objekti koji se iniciraju pomoću takovih klasa iskazuju dva svojstva:

- **nasljeđivanje i**
- **polimorfizam.**

Postupno ćemo upoznati tehniku programiranja kojom se realizira OOP.

Nasljeđivanje

Nasljeđivanje je tehnika kojom se definiranje neke klase vrši korištenjem definicije postojeće klase koja se naziva *temeljna. klasa*. Tako dobivena klasa se naziva *izvedena klasa*.

Članovi temeljne klase postaju i članovi i izvedene klase.

Sintaksa deklaracije izvedene klase najčešće se koristi u obliku:

```
class ime_izvedene_klase : public ime_temeljne_klase
{

  // sadrži članove koji su definirani u temeljnoj klasi
  // definiranje dodatnih članova klase

}
```

Primjerice, neka postoji klasa imena `Temelj` i pomoću nje deklarirajmo klasu `Izveden`

```
class Temelj {
public;
    Temelj() { elem0=0; }
    int elem0;
}
class Izveden : public Temelj {
public:
    Izvedena() {elem1 = 0}
    int elem1;
}
```

Ako pomoću nje deklariramo klasu `Izveden` tada objekti ove klase imaju dva člana `elem0` i `elem1`. Promjerice, objektu `x`, deklariranom s:

```
Izveden x;
```

članskim varijablama pristupamo s:

```
x.elem0  = 5;
x.elem1 = 7;
```

Kažemo da je klasa `Izveden` naslijedila član klase `Temelj`, jer je `elem0` deklariran u klasi `Temelj`.

Ukoliko se ne koristi nasljeđivanje, klasu `Izveden` se može zapisati u funkcionalno ekvivalentnom obliku:

```
class Izveden // bez naslijeđivanja
{
    public:
        Izvedena() {elem0 = 0; elem1=0;}
        int elem0;
        int elem1;
}
```

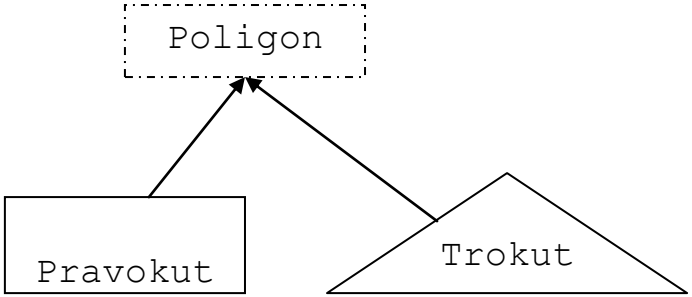
U verziji s nasljeđivanjem konstruktor je zadužen za inicijalizaciju samo onih varijabli koje su deklarirane u pojedinoj klasi. U C++ jeziku vrijedi pravilo :

- **Konstruktor i destruktor se ne naslijeđuje.**
- **Pri inicijalizaciji objekta izvršavaju se svi konstruktori iz hijerarhije naslijeđivanja – najprije konstruktor temeljne klase, a zatim konstruktor izvedenih klasa.**
- **Isto vrijedi i za destruktor, jedino se poziv destruktora vrši obrnutim redoslijedom – najprije se izvršava destruktor izvedene klase, a potom destruktor temeljne klase.**

> ***Napomena*: često se temeljna klasa naziva *superklasa*, a izvedene klase se nazivaju *subklase*. Također se za temeljnu klasu koristi naziv *roditelj* (parent class), a izvedene klase se nazivaju *djeca* (child classes)**

15.1.2 Kada koristimo nasljeđivanje

Nasljeđivanje je korisno u slučaju kada se pomoću temeljne klase može izvesti više klasa. Primjerice klasa `Poligon` može biti temeljna klasa za definiciju klasa `Pravokutnik` i `Trokut`.



U klasi Poligon možemo deklarirati članove koji su zajednički za klasu `Trokut` i `Pravokutnik`. To su `širina` i `visina` poligona.

U klasama `Pravokutnik` i `Trokut`, koje su izvedene klase, definirat ćemo funkciju kojom se računa površina poligona – `Povrsina()`.

```
class Poligon {
  protected:
    int sirina, visina;
  public:
    void init (int a, int b) { sirina=a; visina=b;}
    int Sirina()             {return sirina;}
    int Visina()             {return visina;}
  };

class Pravokutnik: public Poligon {
  public:
    int Povrsina (void)      { return (sirina * visina); }
  };

class Trokut: public Poligon {
  public:
    int Povrsina (void)      { return (sirina * visina / 2); }
  };

int main () {
  Pravokutnik pr;    Trokut tr;
  pr.init (4,5);     tr.init (4,5);
  cout << pr.Povrsina() << endl <<  tr.Povrsina() << endl;
  return 0;
}
```

```
Rezultat izvršenja je:
20
10
```

15.1.3 Public, private i protected specifikatori

Specifikatori `public`, `private` i `protected` određuju pristup članovima klase:

Pristup članovima	public	protected	private
iz temeljne klase	da	da	da
iz prijatelja klase	da	da	da
iz izvedene klase	da	da	ne
izvan klase	da	ne	ne

Razlika specifikatora *private* i *protected* je u tome što se protected članovi mogu koristiti u izvedenim klasama, a *private* članovi se mogu koristiti samo u klasi u kojoj su deklarirani.

Ključne riječi `public`, `private` i `protected` se koriste i kao specifikatori tipa naslijeđivanja.

U našem primjeru, klase Pravokutnik i Trokut su deklarirane s oznakom naslijeđivanja tipa `public`, tj.

```
class Pravokutnik: public Poligon;
class Trokut: public Poligon;
```

Riječ `public` označava da specifikatori pristupa varijablama iz temeljne klase vrijede i u izvedenim klasama. Pravo pristupa prema tipu naslijeđivanja prikazano je u slijedećoj tablici:

Tip nasljeđivanja	Pravo pristupa u temeljnoj klasi		
	`public`	`protected`	`private`

`public`
`protected`
`private`
	`public`
	`protected`
	`private`
		`protected`
		`protetected`
		`private`
			`private`
			`private`
			`private`

Ako se nasljeđivanje specificira s `protected` ili `private`, tada se ne može javno pristupiti članovima koji su temeljnoj klasi deklarirani kao public.Ukoliko se ne označi specifikator naslijeđivanja, podrazumjeva se da izvedena klasa ima `private` specifikator naslijeđivanja.

Što se sve nasljeđuje iz temeljne klase

U pravilu, iz temeljne se klase nasljeđuju sve članske funkcije i varijable osim:

- **konstruktora i destruktora**
- **operatora =**
- **prijateljskih funkcija i klasa**

Uvijek se prije izvršenja konstruktora izvedene klase izvršava predodređeni konstruktor temeljne klase. Ako želimo da se izvrši neki drugi konstruktor, tada to treba eksplicitno zadati u izvedenim klasama u obliku:

```
ime_izvedene_klase(parameteri) : ime_temeljne_klase(parameteri)
{
  ...
}
```

Primjerice naslijeđivanja:

```
class Otac {
  public:
    Otac()       { cout << "otac: konstruktor bez argumenta\n"; }
    Otac(int a)  { cout << "otac: konstruktor s argumentom\n"; }
};

class Kcer : public Otac {
  public:   Kcer(int a) {cout << "kcer: konstruktor s arg\n\n";}
};

class Sin : public Otac {
  public:  Sin(int a) : Otac (a) { cout << "sin: konstruktor s
arg\n\n"; }
};

int main () {
  Kcer ana(1);                  // otac: konstruktor bez argumenta
                                // kcer: konstruktor s argumentom
  Sin anton(1);                 // otac: konstruktor s argumentom
                                // sin: konstruktor s argumentom
  return 0;
}
```

Nadređenje članskih funkcija

U izvedenoj se klasi može definirati članska funkcija istog imena kao i u temeljnoj klasi. Potrebno je razlikovati dva slučaja:

1. **kada obje funkcije imaju iste parametre – tada nastaje nadređenje funkcije (overriding)**
2. **kada funkcije imaju različite parametre – tada nastaje preopterećenje funkcije (overloading).**

Efekte preopterećenja funkcije smo već upoznali i vidjeli da preopterećene funkcije kompajler tretira kao različite funkcije.

Razmotrimo primjer nadređenih funkcija. Uzet ćemo banalni primjer, da se definira klasa `Romb` pomoću klase `Pravokutnik`. Pošto za površinu pravokutnika i romba vrijedi ista zakonitost, klasa `Romb` je definirana već samim nasljeđivanjem

```
class Romb: public Pravokutnik
{
};
```

Ako bi izvršili program:

```
int main () {
  Pravokutnik pr;
  Romb romb;
  pr.init (4,5);
  romb.init (4,5);
  cout << pr.Povrsina() << endl;
  cout << romb.Povrsina() << endl;
  return 0;
}
```

dobili bi isti rezultat za površinu romba i pravokutnika.

Da bi pokazali efekt nadređenja funkcija u klasi Romb ćemo sada definirati funkciju kojom se računa površina koristeći funkciju za površinu iz temeljne klase:

```
class Romb: public Pravokutnik {
 public:
    int Povrsina (void)
     { return Pravokutnik::Povrsina(); }
};
```

Uočimo da se poziv funkcije iz temeljne klase označava imenom temeljne klase i rezolucijskim operatorom `::`.

U slijedećem programu testira se klasa Romb i ujedno pokazuje da se može pristupiti nadređenim javnim funkcijama,

```
int main ()
{
  Romb romb;
  romb.init (4,5);
  cout << romb.Povrsina() << endl;
  cout << romb.Pravokutnik::Povrsina() << endl;
  return 0;
}
```

Nasljeđivanje generičkih klasa

Nasljeđivanje se može koristiti i kod definicije generičkih klasa. Opći oblik deklaracije naslijeđivanja kod generičkih klasa je

```
template <class T>
class izvedena_klasa: public temeljna_klasa<T>
{
  // sadrži članove koji su u temeljnoj klasi
  // definirani pomoću generičkog tipa T
  // definiranje dodatnih članova klase
}
```

Primjer:
Klasu `List`, definiranu u prethofnom poglavlju koristit ćemo kao temeljnu klasu u izvođenju klase `List1`. U Izvedenoj klasi ćemo definirati člansku funkciju `concat()` – služi za spajanje dviju lista

```
#include "list.h"

template <class Type>
class List1: public List<Type>
{
public:
  void    concat(List<Type> & L);
};


template <class Type>  void List1<Type>::concat(List<Type> &otherList)
{
    ListElem *pElem = otherList.First;

    while (pElem != NULL)
       {
         push_back(pElem->Elem);
         pElem = pElem->Next;
       }
}
```

Višestruko nasljeđivanje

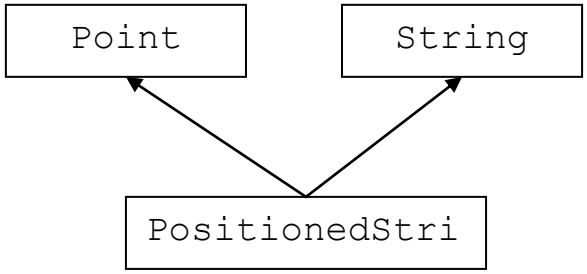
U C++ je dozvoljeno da se nasljeđivanje može realizirati iz više temeljnih klasa. Takovo nasljeđivanje se naziva višestruko nasljeđivanje. Sintaksa višestrukog nasljeđivanja je:

```
class ime_izvedene_klase : lista_temeljnih_klasa
{
  // sadrži članove koji su definirani u temeljnim klasama
  // definiranje dodatnih članova klase
 }
```

Kao primjer uzmimo da u programu koji se odvija u grafičkoj okolini treba ispisati string u točki kojoj su koordinate x,y.

```
class Point
{
protected:
    int m_x;
    int m_y;
public:
    Point() : m_x(0), m_y(0) {}
    void SetPosition(int x, int y) {m_x=xy; m_y=y;}
}
```

Možemo definirati klasu PositionedString koja naslijeđuje svojstva od klase string i klase Point.



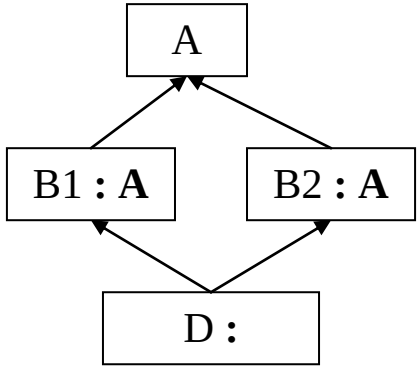

```
class PositionedString: public string, public Point {
//------------
public:
    Draw();
}
```

Objekt klase PositionedString se ponaša kao objekt klase Point, primjerice možemo ga pomicati

```
PositionedString pstr;
pstr = "Hello";              // koristimo kao obicni string
pstr.SetPosition(2, 10);     // pomičemo string
pstr.Draw();
...
```

PROBLEM VIŠESTRUKOG NASLIJEĐIVANJA

Iako višestruko nasljeđivanje može biti vrlo korisno, u novijim programski jezicima (Java, C#) nije ni implementirano. Razloga su dva:
Prvi je razlog da se njime uspostavlja dosta kompliciranа hijerarhija u naslijeđivanju, a drugi razlog je da često može nastati konflikt koji jer prikazan slijedećom slikom:



Klasa D naslijeđuje klase B1 i B1, koje imaju istu temeljnu klasu A.

Pošto se konstruktori ne naslijeđuju vidimo da bi se konstruktor A trebao inicirati dva puta, prvi put unutar konstruktora B1 a drugi put unutar konstruktora B2 (redoslijed poziva konstruktora određen je redoslijedom u specifikaciji naslijeđivanja).
Ovaj se problem može riješiti korištenjem tzv. virtualnih temeljnih konstruktora, ali to ovdje neće biti objašnjeno.

Savjet: Višestruko nasljeđivanje mnogi programeri zovu "goto of the 90's".
Ne preporučuje se njegova upotreba.

Polimorfizam

Polimorfizam je svojstvo promjenjljivosti oblika. Kaže se da je polimorfan onaj program koji je napisan tako da se programski algoritami ili funkcije mogu primijeniti na objekte različitih oblika. U tu svrhum, u C++ jeziku je implementiran mehanizam virtuelnih funkcija.

Najprije ćemo pokazati kako se prilagođenje objektu može dijelom izvršiti već prilikom kompajliranja programa (tzv- statičko povezivanje s objektom – static binding), a zatim ćemo pokazati kao se pomoću pokazivača i virtuelnih funkcija povezivanje s objektom vrši tijekom izvršenja programa (tzv. izvršno povezivanje – run-time binding).

15.2.1. "Is a" odnos objekata

U praksi se skoro isključivo koristi public nasljeđivanje. Razlog tome je što se njime omogućuje tzv. "Is a" odnos objekata iz hijerarhije naslijeđivanja. Možemo kazati

Klasa `Trokut` "je od vrste" klase `Poligon`, i (eng. "kind-of" class relationship)

Objekt klase `Trokut` "je" objekt klase `Poligon` (eng. "is-a" object relationship)

jer se članovima objekta `Trokut` može pristupati kao da se radi o objektu klase `Poligon`.

Ovo "is-a" svojstvo ima programsku implikaciju da se pokazivači i reference, koji mogu biti deklarirani i kao argumenti funkcija, a koji se deklariraju pomoću temeljne klase, mogu koristiti i za manipuliranje s objektima izvedenih klasa.

To ilustrira primjer:

//.. koristimodefinicije klasa Poligon, Pravokutnik i Trokut

```
void IspisDimenzija(Poligon & p)
{
    cout << "sirina = " << p.Sirina() << endl;
    cout << "visina = " << p.Visina() << endl;
}
```

```
int main ()
{
  Pravokutnik pr;
  Trokut tr;
  pr.init (4,5);
  tr.init (4,5);
  cout << "Pravokutnik:" << endl;
  IspisDimenzija(pr);
  cout << "povrsina =" << pr.Povrsina() << endl;
  cout << "Trokut:" << endl;
  IspisDimenzija(tr);
  cout << "povrsina =" << tr.Povrsina() << endl;
  return 0;
}
```

Rezultat izvršenja je:

```
Pravokutnik:
sirina = 4
visina = 5
povrsina =20
Trokut:
sirina = 4
visina = 5
povrsina =10
```

Funkcija `IspisDimenzija()` je definirana s parametrom koji označava referencu objekta temeljne klase Poligon. Pri pozivu ove funkcije stvarni argument funkcije su objekti izvedenih klasa.

Četiri standardne pretvorbe su moguće između objekate izvedene i temeljne javne klase:

1. **Objekt izvedene klase može se implicitno pretvoriti u objekt javne temeljne klase.**
2. **Referenca na objekt izvedene klase može se implicitno pretvoriti u referencu objekta javne temeljne klase.**
3. **Pokazivač na objekt izvedene klase može se implicitno pretvoriti u pokazivač objekta javne temeljne klase.**
4. **Pokazivač na člana objekta izvedene klase može se implicitno pretvoriti u pokazivač člana objekta javne temeljne klase.**

U sljedećem primjeru pokazano je kako se pomoću pokazivača na temeljnu klasu pristupa objektima izvedenih klasa.

```
int main ()
{
  Pravokutnik pravokutnik;
  Trokut trokut;
  Poligon * pPol1 = &pravokutnik;
  Poligon * pPol2 = &trokut;
  pPol1->init (4,5);
  pPol2->init (4,5);
  cout << pravokutnik.Povrsina() << endl;
  cout << trokut.Povrsina() << endl;
  return 0;
}
```

Rezultat:

```
20
10
```

15.2.2. Virtuelne članske funkcije

U oba prethodna primjera samo je djelomično iskazan princip polimorfizma, naime korišten je samo za pristup članskoj funkciji `init`, koja je definirana u temeljnoj i izvedenim klasama. Da bi princip polimorfizma mogli potpuno koristiti potreban je mehanizam kojim bi omogućio i poziv fukcije `Povrsina`. To nije bilo moguće ostvariti jer ta funkcija nije definirana u temeljnoj klasi `Poligon`.

Ipak postoji, mehanizam da se ta funkcija može, makar virtuelno, definirati u temeljnoj klasi . To se postiže tako da se u temeljnoj klasi deklarira funkcija `Povrsina` s prefiksom `virtual`.

```
class Poligon
{
  protected:
    int sirina, visina;

  public:
    void init (int a, int b)    { sirina=a; visina=b; }
    int Sirina() {return sirina;}
    int Visina() {return visina;}

     virtual int Povrsina (void) { return (0); }

  };
```

```
class Pravokutnik: public Poligon {
  public:
    int Povrsina (void)
      { return (sirina * visina); }
 };


class Trokut: public Poligon {
  public:
    int Povrsina (void)
      { return (sirina * visina / 2); }
  };

int main ()
{
  Pravokutnik pravokutnik;
  Trokut trokut;
  Poligon * pPol1 = &pravokutnik;
  Poligon * pPol2 = &trokut;
  pPol1->init (4,5);
  pPol2->init (4,5);
  cout << pPol1->Povrsina() << endl;
  cout << pPol2->Povrsina() << endl;
  return 0;
}
```

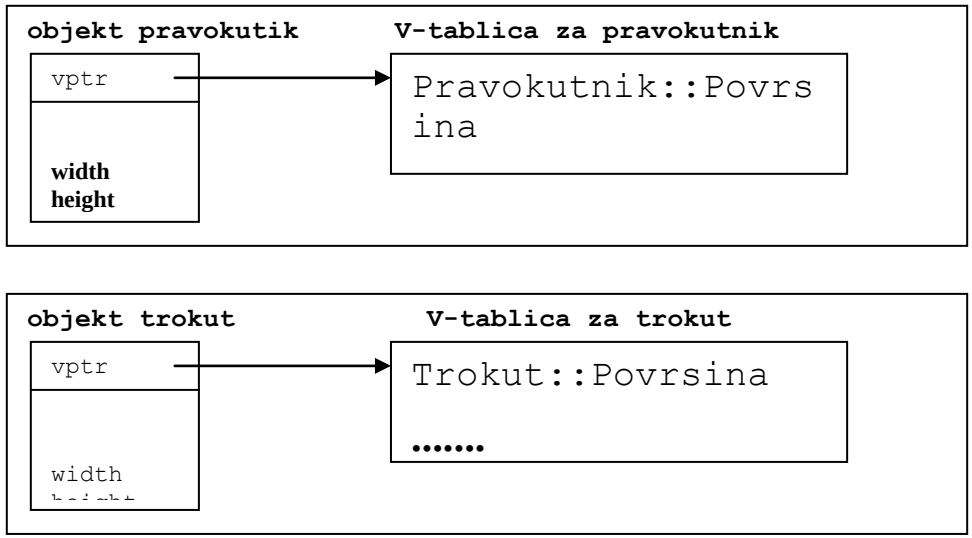
Vidimo da je sada pomoću pokazivača moguće pristupiti svim funkcijama neke klase.

Postavlja se pitanje: na koji način je prepoznato koja funkcija treba biti pozvana.

Da bi to shvatili treba znati slijedeće:

- **kada se u temeljnoj klasi neka funkcija označi kao virtuelana tada se i sve funkcije istog imena u izvedenim klasama tretiraju kao virtuelne funkcije.**
- **u izvedenim se klasama ispred imena virtuelne funkcije može napisati specifikator virtual, iako to nije nužno.**
- **poziv virtuelnih funkcija vrši se drukčije nego poziv regularnih članskih funkcija.**

Za svaki objekt koji ima virtuelne funkcije kompajler generira posebnu tablicu (V-tablicu) u koju upisuje adresu virtuelnih funkcija, također uz članove klase zapisuje i pokazivač na ovu tablicu (vptr). To prikazuje slijedeća slika:



Objasnimo način na koji se prepoznaje i vrši poziv virtuelne funkcije:

1. **pretpostavimo da je adresa nekog objekta pridjeljena nekom pokazivaču (ili je referenca)**
2. **kada se treba izvršiti poziv virtuelne funkcije najprije se dobavlja adresa tablice pokazivača virtuelnih funkcija koja je zapisana u pokazivaču `vptr`.**
3. **zatim se iz tablice dobavlja adresa te funkcije i konačno indirekcijom tog pokazivača se vrši poziv funkcije.**

Iz ovog objašnjenja proizilazi da je izvršenje programa s virtuelnim funkcijama sporije nego izvršenje programa s regularnim funkcijama, jer se gubi vrijeme za dobavu adrese virtuelne funkcije. Bez obzira na ovu činjenicu rad s virtuelnim funkcijama je od velike koristi jer se pomoću njih postiže potpuni polimorfizam, a to je najvažniji element objektno orjentiranog programiranja.

> **Zapamti: virtuelne funkcije nam služe za potpunu primjenu polimorfizma. To omogućuje da se programska rješenja lako prilagođuju različitim objektima iz hijerarhije naslijeđivanja.**

15.2.3. Apstraktne temeljne klase

Apstraktne temeljne klase su klase u kojima je definirana bar jedna čista virtuelna funkcija.

Čista virtuelna funkcija se označava tako da se iza deklaracije funkcije napiše = 0, tj

Sintaksa čiste virtuelne funkcije:
virtual *deklaracija_funkcije* = 0;

Klase koje sadrže čiste virtuelne funkcije ne mogu se koristiti za deklaraciju objekata, ali se pomoću njih može deklarirati pokazivač na objekte ili argument funkcije koji je referenca objekta.
Primjerice, klasa Poligon se može tretirati kao apstraktna temeljna klasa :

```
// abstract class Poligon
 class Poligon {
  protected:
    int sirina, visina;
  public:
    void init (int a, int b)
      { sirina=a; visina=b; }
    virtual int Povrsina (void) =0;
  };
```

jer nije predviđeno da se njome deklarira statičke objekte.

Napomena: **Kompajler ne generira izvršni kod za čiste virtuelne funkcije, stoga u svim klasama koje se izvode iz apstraktne temeljne klase mora biti implementirana ta funkcija.**

Pregledom prethodnih programa vidi se da se može pristupiti svim objektima iz hijerarhije nasljeđivanja pomoću pokazivača na temeljnu klasu (`Poligon` *). To znači da ako se u temeljnoj klasi definira funkcije koje koriste virtuelne funkcije, tada te funkcije mogu vrijediti za sve objekte iz hijerarhije nasljeđivanja.

Primjerice, u klasi Poligon ćemo definirati funkciju `PrintPovrsina`() kojom se ispisuje vrijednost površine. To je pokazano u programu inherit7.cpp.

```
class Poligon {
  protected:
    int sirina, visina;
  public:
    void init (int a, int b)    { sirina=a; visina=b; }
    int Sirina() {return sirina;}
    int Visina() {return visina;}
    virtual int Povrsina (void) = 0;
    void PrintPovrsina (void)
      { cout << this->Povrsina() << endl; }

  };
```

```
class Pravokutnik: public Poligon {
  public:
    int Povrsina (void) { return (sirina * visina); }
  };

class Trokut: public Poligon {
  public:
    int Povrsina (void) { return (sirina * visina / 2); }
  };

int main ()
{
  Pravokutnik pravokutnik;
  Trokut trokut;
  Poligon * pPol1 = &pravokutnik;
  Poligon * pPol2 = &trokut;
  pPol1->init (4,5);
  pPol2->init (4,5);
  pPol1->PrintPovrsina();
  pPol2->PrintPovrsina();
  return 0;
}
```

Uočite da je u funkciji

```
void PrintPovrsina (void)
      { cout << this->Povrsina() << endl; }
```

pristup funkciji `Povrsina` je izvršen pomoću `this` pokazivača. To osigurava da će biti pozvana funkcija `Povrsina` koja pripada aktivnom objektu, jer `this` pokazivač uvijek pokazuje na aktivni objekt.

15.2.4. Virtualni destruktor

Prethodni program smo mogli napisati i u sljedećem obliku:

```
int main ()
{
  Poligon * pPol = new Pravokutnik;
  pPol->init(4,5);
  pPol->PrintPovrsina();
  delete pPol;              // dealociraj memoriju Pravokutnika

   pPol = new Trokut;       // ponovo koristi pokazivač
  pPol->init(4,5);
  pPol->PrintPovrsina();
  delete pPol;
   return 0;
}
```

U prvoj liniji se alocira memorija za objekt tipa pravokutnik. Tom se objektu dalje pristupa pomoću pokazivača na temeljni objekt tipa Poligon. Kada se obave radnje s ovim objektom oslobađa se zauzeta memorija. Zatim se isti pokazivač koristi za rad s objektom tipa Trokut.
Na prvi pogled sve izgleda uredu, i većina kompajlera će izvršiti ovaj program bez greške. Ipak, ako bi ovaj program uzeli kao obrazac za za rad s dinamičkim objektima, onda se u njemu krije jedna ozbiljna greška, a to je da se neće se izvršiti poziv destruktora. Zašto?

Da bi odgovorili na ovo pitanje prisjetimo se što se događa kad se pozove operator delete.

Tada se najprije poziva destruktor objekta i destruktori svih klasa koje on nasljeđuje, a zatim se vrši dealociranje memorije. U ovom slučaju pokazivač `pPol` je deklariran kao pokazivač na temeljni objekt tipa `Poligon`, pa se neće izvršiti poziv destruktora `Pravokutnika` i `Trokuta` (u ovom programu to nema neke posljedice, jer u klasama `Pravokutnik` i `Trokut` destruktor nema nikakovi učinak).

<u>Da bi se omogućilo da se može pozvati destruktor izvedenih klasa pomoću pokazivača na temeljnu klasu potrebno je da se destruktor temeljne klase deklarira kao virtuelna funkcija.</u>

Redoslijed poziva konstruktora i destruktora se može analizirati pomoću programa inherit8.cpp.

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

class Superclass {
public:
    Superclass (){cout << "Konstruktor temeljne klase\n"; }
    virtual ~Superclass(){cout << "Destruktor temeljne klase\n";}
};

class Subclass : public Superclass
{
public:
    Subclass () {cout << "Konstruktor izvedene klase\n"; }
    ~Subclass() {cout << "Destruktor izvedene klase\n";}
};
```

```
int main ()
{
    Superclass * p = new Subclass;
    delete p;
    return 0;
}
```

Dobije se ispis:

```
Konstruktor temeljne klase
Konstruktor izvedene klase
Destruktor izvedene klase
Destruktor temeljne klase
```

Zadatak: Provjerite, ako se u temeljnoj klase destruktor deklarira bez prefiksa `virtual`, dobije se ispis:

```
Konstruktor temeljne klase
Konstruktor izvedene klase
Destruktor temeljne klase
```

Dakle, u ovom se slučaju ne poziva destruktor izvedene klase.

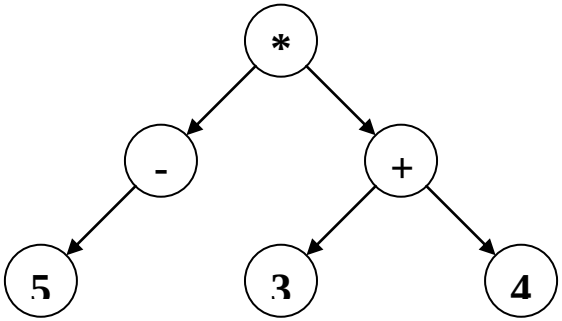
Zapamtite: Uvijek je korisno deklarirati funkcije temeljne klase kao virtuelne funkcije. Time se omogućuje polimorfizam klasa. Destruktor temeljne klase treba deklarirati kao virtuelnu funkcija.
Kada se kasnije u radu pokaže da neku člansku funkciju nije potrebno koristiti polimorfno, tada se funkcija može deklarirati kao nevirtuelna članska funkcija, jer se time dobija nešto brže izvršenje poziva funkcija.

15.2.5. Polimorfizam na djelu – izvršenje aritmetičkih izraza

Pokazat ćemo nešto kompliciraniji primjer: program kojim se računaju aritmetički izrazi. Analizirajmo najprije strukturu aritmetičkih izraza. Oni se sastoje od više članova i faktora koji mogu biti grupirani unutar zagrada. Članovi i faktori sadrže operande i operatore, a izraz u zagradama se tretira kao jedinstveni operand.

Izrazi oblika -3 ili +7 se nazivaju unarni izrazi, a izrazi oblika 3 + 7 ili 6.7 / 2.0 se nazivaju binarni izrazi. Oni se jednostavno računaju tako da se na operande primijeni računska operacija definirana zadanim operatorom.

U slučaju kompleksnijih izraza, primjerice
-5 * (3+4)
potrebno je prethodno izraz na adekvatan način zapisati u memoriji, s točno označenim redoslijedom izvršenja operacija, kako bi se pravilno primijenilo pravilo djelovanja asocijativnosti i prioriteta djelovanja operatora.
Kod modernih kompajlera i intepretera za internu prezentaciju izraza koristi se zapisi u obliku razgranate strukture koje se naziva apstraktno sintaktičko stablo. Primjerice, gornji izraz se može apstraktno predstaviti u obliku:



Slika 15.5. Apstraktno sintaktičko stablo izraza: -5 * (3+4)

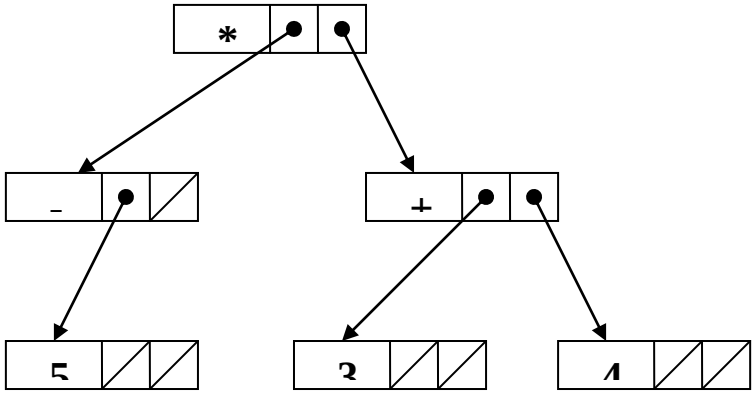
Stablo ima više čvorova i grana. U čvorovima su zapisani sintaktčiki entiteti: operatori i operandi.

Čvor iz kojeg započima grananje stabla naziva se korijen stabla. Čvorovi koji nemaju grane nazivaju se lišće stabla. Svi ostali čvorovi se nazivaju unutarnji čvorovi. Unutarnji čvorovi i lišće su podstabla stabla koje je definirano korijenom stabla.

Apstraktno sintaktičko stablo aritmetičkih izraza se može izgraditi na slijedeći način:

U korijen stabla se zapisuje operator najvećeg prioriteta. Zatim se crtaju dvije grane koje povezuju čvor lijevog i desnog operanda. Ako su ti operandi ponovo neki izrazi, proces izgradnje se nastavlja tako da se u te čvorove ponovo upisuje operator najvećeg prioriteta u podizrazima. Ako su operandi brojevi, u čvor se upisuje vrijednost broja. Proces izgradnje završava tako da se u lišću nalaze brojevi, a u višim čvorovima se nalaze operatori.

U programu se sintaktičko stablo može zapisati pomoću struktura koje sadrže informaciju čvora i pokazivače na čvor



Slika 15.6. Realizacija apstraktnog sintaktičkog stabla izraza: -5 * (3+4)

Vidimo da svi čvorovi ne sadrže istu informaciju: u unutarnjim čvorovima je sadržana informacija o operatorima, a čvorovi lišća sadrže operande.

Čvorovi koji sadrže binarne operatore trebaju imati dva pokazivača za vezu s lijevim i desnim operandom, čvorovi koji sadrže unarni operator trebaju samo jedan pokazivač, a čvorovi lišća ne trebaju ni jedan pokazivač.

Očito je da čvorove treba tretirati polimorfno. Moguća su različita rješenja.

Pokazat ćemo rješenje u kojem se za sve tipove čvorova koristi temeljna virtuelna klasa ExprNode, a pomoću nje ćemo definirati klase BinaryExprNode, UnaryExprNode i NumNode.

```
class ExprNode  // temeljna klasa za sve tipove izraza
{
   friend ostream& operator<< (ostream&, const ExpNode *);
public:
    ExpNode() {}
    virtual ~ExpNode() { }
    virtual void print (ostream&) const = 0;
    virtual double execute() const = 0;
};

ostream&   operator<<(ostream& out, const ExpNode *p)
{
    p->print(out); // virtualni poziv, vrijedi za sve  subklase
    return out;
}
```

U temeljnoj klasi ExprNode definirane su dvije čiste virtuelne funkcije print() i execute(). Funkcija print() služit će za ispis sadržaja stabla, a funkcija execute() će izvršiti aritmetički izraz koji je definiran podstablom. Operator << omogućuje da se ispiše sadržaj stabla iz poznatog pokazivača na stablo (ili podstablo).

NumNode

Klasa NumNode se izvodi iz ExprNode. Ona sadrži numeričke operande tipa double. Funkcija print() ispisuje numeričku vrijednost, a funkcija execute() vraća tu vrijednost, jer je vrijednost aritmetičkog izraza koji sadrži samo jedan broj upravo vrijednost tog broja.

```
class NumNode: public ExprNode
{
    double n;
public:
    NumNode (double x): n (x) { }
    ~NumNode() { };
    void print (ostream& out) const { out << n; }
    double execute() const {return n;}
};
```

UnaryExprNode

Klasa UnaryExprNode se izvodi iz ExprNode. Ona sadrži unarni operator + ili -, te pokazivača na čvor koji je operand.

```
class UnaryExprNode: public ExprNode
{
    const int op;          // tip operatora ('+' ili '-')
    ExprNode * oprnd;      // pokazivac na operanda
public:
    UnaryExprNode (const int a,  ExprNode * b): op (a), oprnd (b)
       { }
    ~UnaryExprNode() {delete oprnd;}
    double execute() const
      {  return (op=='-')? -oprnd->execute() : oprnd->execute();}
    void print (ostream& out) const
      {  out << "(" << (char)op << oprnd << ")";}
};
```

BinaryExprNode

Klasa BinaryExprNode se također izvodi iz ExprNode. Ona sadrži binarni operator (+, -, * ili /) te pokazivače na čvorove koji je predstavljaju lijevi i desni operand.

```
class BinaryExprNode: public ExprNode {
private:
    const int op;  // tip operatora ('+',  '-', '*' ili '/')
    ExprNode * left;   // pokazivac na lijevi operand
    ExprNode * right;  // pokazivac na desni operand
public:
    BinaryExprNode (const int a, ExprNode *b, ExprNode *c):
        op (a), left (b), right (c) { }
    ~BinaryExprNode() {delete left; delete right;}
    double execute() const;
    void print (ostream& out) const
    {  out << "(" << left << (char)op << right << ")"; }
};

double BinaryExprNode:: execute()  const {
    switch(op){
    case '+': return left->execute() + right->execute();
    case '-': return left->execute() - right->execute ();
    case '*': return left->execute() * right->execute();
    case '/': // provjeri dijeljenje s nulom
        {  double val = right->execute();
        if(val != 0)
            return left-> execute() / val;
        else
                return 0;
```

```
            }
            default: return 0;
        }
    }
```

U ovom slučaju funkcija print() ispisuje, unutar zagrada, lijevi operand, operator i desni operand. Funkcija execute() vraća vrijednost koja se dobije primjenom operatora na lijevi i desni operand. Posebno je analiziran slučaj operacije dijeljenja koko bi se izbjeglo dijeljenje s nulom. Konstruktor formira čvor tako da prima argumente: operator, i pokazivač na čvorove desnog i lijevog operanda. Destruktor dealocira memoriju koju zauzimaju operandi.

Testiranje provodimo sljedećom main() funkcijom:

```
int   main()
{// formiraj stablo za izraz -5 * (3+4)

    ExprNode* t= new BinaryExprNode('*',
                 new UnaryExprNode ('-',
                                    new NumNode(5)),
                 new BinaryExprNode('+',
                                    new NumNode(3),
                                    new NumNode(4)));
    // ispiši i izvrši izraz
    cout << t << "=" << t-> execute() << "\n";

    // dealociraj memoriju
    delete t;
    return 0;
}
```

Uočimo da se svi čvorovi alociraju dinamički. Počima se od korijena i zatim se dodaju podstabla.

Kada se ovaj program izvrši, dobije se ispis:

```
((-5)*(3+4))=-35
```

Ovaj primjer pokazuje da se korištenjem temeljne virtuelne klase može pomoću pokazivača temeljne klase manipulirati sa svim objektima iz izvedenih klasa. To je, bez sumnje, najvažniji mehanizam objektno orijentiranog programiranja.

<u>Zadatak:</u> Prethodni primjer realizirajte tako da članska funkcija print() ispisuje izraz u obrnutoj poljskoj notaciji (postfiks oblik).

Pomoć: Dovoljno je da se u u klasi BinaryExprNode članska funkcija

```
void print (ostream& out) const
{  out << "(" << left << (char)op << right << ")"; }
```

napiše u obliku:

```
void print (ostream& out) const
{  out << left << " "<< right << " "<< (char)op << ""; }
```

Tada će se dobiti ispis:

```
(-5) 3 4 + * =-35
```

Ovime smo pokazali da se prethodno sintaktičko stablo lako možemo koristiti ne samo kao intepreter, već i kao kompajler. U ovom slučaju se infiks izrazi kompajliraju u postfiks izraze.

- .-

Ovdje su prikazani elementi objektno orjentiranog programiranja na jednostanim primjerima. U praksi se objektno orjentirano programiranje koristi u veoma složenim softverskim projektima.

Kasnije ćemo pokušati dati odgovor na slijedeće pitanja:

- **Kako pristupiti analizi i izradi složenih programa?**
- **Postoje li obrasci po kojima se može riješiti neke tipične probleme?**
- **Koji softverski alati su pogodni za razvoj OOP programa**