OBJEKTNO PROGRAMIRANJE 1

Oznaka predmeta: OOP

Predavanje broj: 11

Nastavna jedinica: Preklapanje operatora. Izvođenje.

Nastavne teme:

Operator []. Operator ->. Operatori new i delete. Operator =. Operatori konverzije tipova. Klase istream, ostream, ifstream, ofstream. Ulazno/izlazne operacije za korisničke tipove. Definisanje izvedene klase u jeziku C++. Privatno, zaštićeno i javno izvođenje. Prava pristupa. Semantička razlika između privatnog javnog izvođenja. Izvedene klase i konverzije. Konstruktori i destruktori izvedenih klasa. Realizacija izvedenih klasa. Višestruko izvođenje. Višestruki podobjekti. Virtuelne osnovne klase. Inicijalizacija osnovnih klasa. Konverzije pokazivača na izvedenu klasu u pokazivač na osnovnu klasu. Polimorfizam. Virtuelne funkcije. Dinamičko vezivanje. Virtuelne funkcije i virtuelne osnovne klase

Predavač: prof. dr Perica S. Štrbac, dipl. ing.

Literatura:

Dragan Milićev, “Objektno orijentisano programiranje na jeziku C++”, Mikro knjiga, Beograd, 2005.

Operator []

- Operacija indeksiranja oblika: *izraz1*[*izraz2*] smatra se binarnom operacijom.
  - Operator je [], prvi operand je *izraz1*, a drugi operand je *izraz2*.
- Ova operatorska funkcija mora biti nestatička funkcija članica neke klase X.
  - Ako je x objekat klase X, izraz oblika x[*izraz*] se tumači kao poziv operatorske funkcije članice: x.operator[](*izraz*).
- U sledećem primeru pomoću operatora [] pristupa se elementu vektora koji sadrži int elemente.
  - Povratna vrednost je referenca na int tako da je omogućeno korišćenje lvalue svojstva. Dodat je i poziv atexit korisničke funkcije pri korišćenju funkcije exit za slučaj korišćenja nedozvoljenog indeksa pri pristupu elementu vektora.

```
#pragma once
const int SIZE = 10;
class Vektor{
   private:
      int v[SIZE];
   public:
      Vektor();
      int& operator[](int i);
};
```

Operator []

```
#include "Vektor.h"
#include <iostream>
Vektor:: Vektor(){
    register int i;
    for(i = 0; i < SIZE; i++)       v[i] = i;
}
int& Vektor::operator[](int i){
if( ( i>=SIZE ) || ( i<0 ) ) exit(EXIT_FAILURE);
    return v[i];
}
```

```
#include "Vektor.h"
#include <iostream>
using namespace std;
int main(){
   Vektor V;
   cout<< "Value of V[2]   : " << V[2] <<endl;
   int nova = 555;
   V[5] = nova;
   cout<< "Value of V[5]   : " << V[5]   <<endl;
   cout<< "Value of V[200] : " << V[200] <<endl;
   return 0;
}
```

Operator ->

- Unarna operacija posrednog pristupa članu oblika: *izraz1->izraz2* kao operatorska funkcija mora biti nestatička funkcija članica neke klase X i mora da vrati kao rezultat ili pokazivač na klasu koja poseduje član c, ili objekat ili referencu na klasu za koju je opet definisan operator ->.
- Najčešće se njegovo korišćenje odnosi na tzv. "pametne pokazivače" ("*smart pointers*"), koji, osim svoje osnovne namene da ukazuju na objekat, obavljaju i neki dodatni posao.
  - klasa Xptr realizuje "pametni pokazivač" na klasu X, tako da se broji svaki pristup objektu klase X, kao i bilo kom njegovom članu preko pokazivača.

```
class X { public: int m; };
class Xptr {
    X *p;
    int usage;
  public:
    Xptr ( X *px) : p(px), usage(0) {}
    int getUsage()  { return usage;       }
    X& operator*()  { usage++; return *p; }
    X* operator->() { usage++; return  p; }
};
```

Operator ->

```
#include <iostream>
using namespace std;
int main(void)
{
    X x;
    Xptr pobj=&x;      // pobj.p ""ukazuje"" na x;
    (*pobj).m=1;       // poziva se: (pobj.operator*()).m;
    cout<<"(*pobj).m=1; usage="
                         <<pobj.getUsage()<<" x.m="<<x.m<<endl;
    int i=pobj->m;     // poziva se: (pobj.operator->())  ->  m;
    cout<<"i=pobj->m;   usage="
                         <<pobj.getUsage()<<" i="<<i<<endl;
    pobj->m=3;         // i ovde isto;
    cout<<"pobj->m=3;   usage="
                         <<pobj.getUsage()<<" x.m="<<x.m<<endl;
    return 0;
}
```

Operatori new i delete

- Pri kreiranju dinamičkog objekta implicitno se poziva podrazumevana operatorska funkcija sa imenom operator new. Ona obavlja alokaciju potrebnog prostora u dinamičkoj memoriji za smeštanje objekta traženog tipa. Ako ova funkcije ne pronađe potreban prostor u dinamičkoj memoriji, vraća 0.
- Programer može da utiče na ponašanje funkcije *operator new* u slučaju da ona ne pronađe potreban prostor u dinamičkoj memoriji.
- Ova funkcija u tom slučaju poziva funkciju (koja se najčešće naziva "*new-handler*") koja je specificirana poslednjim pozivom funkcije:

```
void (*set_new_handler (   void(*)()   )   )();
```

- Funkcija je deklarisana u standardnom zaglavlju <cnew> i prima kao argument pokazivač na funkciju tipa void() a vraća rezultat istog tipa.
- Poziv ove funkcije vraća pokazivač na funkciju koja je prethodno bila postavljena kao *new-handler*, a postavlja funkciju dostavljenu kao argument za novi *new-handler*.
- Ako postoji funkcija koja je postavljena kao *new-handler*, funkcija *operator new* će pozvati ovu funkciju kada ne uspe da nađe potreban prostor. Programer može obezbediti da postavljena funkcija reši situaciju, umesto da vrati 0.
- Ako programer preuzme kontrolu nad dinamičkom alokacijom objekata neke klase mogu se preklopiti operatori new i delete za neku klasu.

Operatori new i delete

- Funkcija X::operator new klase X je uvek statička funkcija članica, čak i ako nije eksplicitno deklarisana kao static. Poziva se pre nego što se objekat klase X zaista kreira. Prvi argument je celobrojnog tipa size_t (deklarisan u <cstddef>) predstavlja veličinu objekta u jedinicama veličine (sizeof(char)) a ostali argumenti mogu biti proizvoljnog tipa, i može ih biti proizvoljno mnogo (notacija rečena ranije). Ova funkcija vraća tip void*.
- Funkcija X::operator delete klase X je uvek statička funkcija članica, čak i ako nije eksplicitno deklarisana kao static. Poziva se posle ukidanja objekta klase X. Prvi argument mora biti tipa void*, a može da se deklariše i drugi argument tipa size_t. Ova funkcija mora imati povratni tip void.
- Za ove funkcije članice važe uobičajena pravila kontrole prava pristupa.

```
#include <cstddef>
class X {
    //...
 public:
   void* operator new (size_t sz){ return ::operator new(sz); }
                                                  // koristi se ugrađeni new;
   void operator delete (void *p){ ::operator delete(p); }
                                                 // koristi se ugrađeni delete;
   //...
};
```

Operatori new i delete

- Operator new se poziva pri kreiranju dinamičkih objekata klase, neposredno pre poziva odgovarajućeg konstruktora te klase dok se operator delete poziva pri ukidanju objekta, neposredno posle poziva destruktora klase.
- Operatori new i delete ne operišu nad "živim" objektom, već sa "sirovom" memorijom, pre kreiranja, odnosno posle ukidanja objekta (oni obezbeđuju i oslobađaju potreban memorijski proctor).
- Prevodilac može poziv operatora new da ugradi na početak kôda svakog konstruktora gde se ovaj operator poziva samo za dinamičke objekte. Ovaj poziv može biti podrazumevani poziv sa samo jednim argumentom tipa size_t. Zato se preporučuje da se za preklapanje operatora new obezbedi funkcija new koja se može pozvati sa samo jednim argumentom tipa size_t.

```
class X {  //...
  public:
    //...
    void* operator new (size_t s) { return ::operator new(s); }
    void* operator new (size_t, int position);
};
```

- Prevodilac može ugraditi na kraj destruktora poziv operatora delete.
- Funkcije članice X::operator new i X::operator delete ne mogu biti virtuelne, ali se nasleđuju.

Operator =

- Operator dodele = može se preklopiti, definisanjem operatorske funkcije `operator=` za klasu X.
- Ova operatorska funkcija mora biti nestatička funkcija članica neke klase X.
- `Operator=` je jedina operatorska funkcija koja se NE nasleđuje.
- Operatorsku funkciju operator= treba definisati u svim slučajevima kada podrazumevani postupak dodeljivanja (član po član) za neku klasu nije primeren, kad god nije primeren ni postupak inicijalizacije član po član.

```
class X {
  int *pi; // ukazuje na pridruženi dinamički objekat;
public:
  X(int i);                    // konstruktor;
  X(const X& x);               // konstruktor kopije;
  X& operator= (const X&);     // operator dodele;
  ~X() { delete pi; }          // destruktor;
};

X::X(int i)      : pi(new int(  i  )) {}   // konstruktor;

X::X(const X& x) : pi(new int(*x.pi)) {}   // konstruktor kopije;
```

Operator =

```
X& X::operator= (const X& x)          // operator dodele;
{
  if (this!=&x)                       // provera da nije dodela x=x;
  {
   delete pi;  pi=new int(*x.pi);
  }
  return *this;
}

extern X f(X x1);

void g() {
  X xa=3, xb=1;
  X xc=xa;       // poziva se konstruktor kopije;
  xa=f(xb);      // poziva se konstruktor kopije samo za
                 // formalni argument x1 funkcije f,
                 // a za xa se poziva xa.operator=(f(xb));
  xc=xa;         // poziva se xc.operator=(xa);
}
```

Operatori konverzije tipova

- Operatorska funkcija članica klase X oblika: `operator tip`, definiše korisničku konverziju iz tipa X u tip *tip*. Ova konverzija može se vršiti implicitno, na svim mestima na kojima se implicitna konverzija vrši, kao i eksplicitno.

```
class complex {
  double real,imag;
public:
  complex (double r=0, double i=0) : real(r), imag(i) {}
  operator double() { return real; } //konverzija iz complex u double
  //...
};
void func() {
  complex c(3.4,6);
  double d=c;             // implicitna konverzija  complex -> double
  double f=5.0+double(c);// eksplicitna konverzija complex -> double
  f=(double)c-3.0;        // isto kao malopre, notacija C cast
}
```

- U deklaraciji operatorske funkcije za konverziju, *tip* može biti ime bilo kog tipa, i ugrađenog i korisničkog, ili ime korisničkog tipa proširenog rečju class (ili struct, union ili enum) ispred imena, ili tip pokazivača, reference, ili pokazivača na članove klase, i sve to proizvoljno specificirano sa const ili volatile.
- Operatorske funkcije za konverziju se nasleđuju, i mogu biti virtuelne.

Pregled svojstava članova, prijatelja i specijalnih funkcija

	Nasleđuje se	Može biti virtualan	Može imati povratni tip	Član ili prijatelj	Generiše ga prevodilac po potrebi
konstruktor	**NE**	**NE**	**NE**	član	**DA**
destruktor	**NE**	da	**NE**	član	**DA**
konverzija	da	da	**NE**	član	ne
`=`	**NE**	da	da	član	**DA**
`()`	da	da	da	član	ne
`[]`	da	da	da	član	ne
`->`	da	da	da	član	ne
op`=`	da	da	da	bilo šta	ne
`new`	da	**NE**	void*	statički član	ne
`delete`	da	**NE**	void	statički član	ne
ostali operatori	da	da	da	bilo šta	ne
ostali članovi	da	da	da	član	ne
prijatelji	**NE**	**NE**	da	prijatelj	ne

Klase istream i ostream

- Ulazni i/ili izlazni tok (*stream*) je logički koncept koji predstavlja sekvencijalni ulaz ili izlaz znakova na neki uređaj ili datoteku. Tok predstavlja apstrakciju, pa se tokovi u biblioteci realizuju klasama.
- Biblioteka sa deklaracijama u zaglavlju <iostream> sadrži dve osnovne klase, klasu istream i klasu ostream.
- Svakom objektu ovih klasa može da se pridruži jedna datoteka za ulaz/izlaz, tako da se datotekama pristupa isključivo preko ovakvih objekata, odnosno funkcija članica ili prijatelja ovih klasa. Time je podržan princip enkapsulacije.
- U ovoj biblioteci definisana su i dva korisniku dostupna (globalna) statička objekta:
  - objekat cin klase istream koji je pridružen standardnom ulaznom uređaju
  - objekat cout klase ostream koji je pridružen standardnom izlaznom uređaju (obično ekran).
- Klasa istream ima funkciju članicu operator>> za sve ugrađene tipove:

```
istream& istream::operator>> (tip &t); //tip je neki ugrađeni tip
```

- Klasa ostream ima preklopljen operator << za sve ugrađene tipove:

```
ostream& ostream::operator<< (tip x); // tip je neki ugrađeni tip
```

Klase istream i ostream

- Ove funkcije realizuju ulaz, odnosno izlaz svog drugog argumenta, za tok koji je naveden kao prvi operand. Ove funkcije vraćaju reference na isti tok nad kojim je izvršena operacija ulaza/izlaza, tako da se može vršiti višestruki ulaz/izlaz u istom izrazu, višestrukim navođenjem operacija << i >>.
- Ovi operatori grupišu sleva udesno.

```
#include <iostream>  // obavezno ako se želi ulaz/izlaz;
void main () {
  int i;
  std::cin>>i;                    //učitava se i;
  std::cout<<"i="<<i<<std::endl;//ispisuje npr. i=10 i ide u novi red
}
```

- Objekat cout je statički objekat klase ostream, kome je pridružen standardni izlazni uređaj.
- Za klasu ostream je definisan operator <<, koji vrednost desnog operanda, ugrađenog tipa char*, šalje na izlaz, čime objekat cout dovodi u novo stanje.
- Operator << vraća referencu na isti ovaj objekat cout, pa se nad njim primenjuje sledeća operacija <<, sa desnim operandom tipa int. Vrednost ovog operanda se šalje na izlaz, a operacija opet vraća referencu na isti objekat cout.
- Objektna realizacija omogućava definisanje proizvoljno mnogo tokova za proizvoljno mnogo uređaja i datoteka.

Ulazno/izlazne operacije za korisničke tipove

- Korisničke operatorske funkcije << i >> treba da imaju prvi argument tipa istream&, odnosno ostream&, kako bi delovale na objekat koji je prvi operand, te da vraćaju reference na isti taj objekat kako bi se mogle pozivati u nizu.

```
#include <iostream>
using namespace std;
class complex {
    double real,imag;
    friend ostream& operator<< (ostream&,complex&);
  public:
    //...
};
ostream& operator<< (ostream &os, complex &c) {
  return os << "(" << c.real << "," << c.imag << ")";}
void main () {
  complex c(0.5,0.1); cout<<"c="<<c<<'\n'; //ispisuje se: c=(0.5,0.1)
 }
```

- Navedeno ispisivanje kompleksnog broja realizovano je pozivom standardnih operacija izlaza za komponente kompleksnog broja, koje su ugrađenih tipova i za koje postoji definisano značenje operatora << za izlaz.
- Potrebno je da operatorska funkcija prima argument tipa reference na ostream, i da nad tim argumentom izvrši bibliotečnu operaciju izlaza.

Zadatak

```
//_____________________Complex.h
#pragma once
#include <cmath>
#include <iostream>
#include <fstream>
using namespace std;
class Complex{
  private:
    double real;      double imag;
  public:
    Complex(double=0,double=0);
    Complex operator +(Complex);
    Complex operator -(Complex);
    Complex operator *(Complex);
    Complex operator /(Complex);
    Complex operator !(); //operator ! koristice se za dobijanje
                          //konjugovano kompleksnog, za a+bi to je a-bi
    friend ostream&  operator <<(ostream &s,Complex &c);  //za ekran
    friend ofstream& operator <<(ofstream &s,Complex &c); //za fajl
};
```

Zadatak

```
//_____________________Complex.cpp
#include "Complex.h"
Complex::Complex(double real, double imag)
:
real(real), imag(imag)
{}
Complex Complex::operator+(Complex drugi_operand){
  return Complex(this->real + drugi_operand.real,
                 this->imag + drugi_operand.imag) ;
}
Complex Complex::operator-(Complex drugi_operand){
  return Complex(this->real - drugi_operand.real,
                 this->imag - drugi_operand.imag) ;
}
Complex Complex::operator*(Complex drugi_operand){
  return Complex((this->real * drugi_operand.real) -
                 (this->imag * drugi_operand.imag) ,
                 (this->real * drugi_operand.imag) +
                 (this->imag * drugi_operand.real) );}
```

Zadatak

```
//_____________________Complex.cpp
Complex Complex::operator/(Complex drugi_operand){
  double delilac = pow(drugi_operand.real,2) +
                   pow(drugi_operand.imag,2);
  if(0==delilac) {
    cout<<"Delilac Vam je 0 !!!"<<endl;
    cout<<"Pritisnite taster za izlazak iz programa..."<<endl;
    system("pause");
    exit(EXIT_FAILURE);
  }
  return Complex(
    ( (this->real * drugi_operand.real) +
      (this->imag * drugi_operand.imag) ) / delilac,
    ( (this->imag * drugi_operand.real) -
      (this->real * drugi_operand.imag) ) / delilac );
}
Complex Complex::operator!(){
  return Complex(this->real, -this->imag);
}
```

Zadatak

```
//_____________________fajl gde je main
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
using namespace std;
#include "Complex.h"

ostream& operator<<(ostream& os, Complex& c) {
  //OBLIK:  Re + Im * i
  os<<setprecision(2)<<fixed;
  os<<noshowpos<<c.real;
  if(c.imag != 0) os<<showpos  <<c.imag<<"i";
  //os<<"("<<c.real<<","<<c.imag<<")"<<endl; //OBLIK (Re,Im)
  return os;
}
```

Zadatak

```
ofstream& operator<<(ofstream& os, Complex& c) {
  //OBLIK:  Re + Im*i
  os<<setprecision(2)<<fixed;
  os<<noshowpos<<c.real;
  if(c.imag != 0) os<<showpos  <<c.imag<<"i";
  return os;
}
void main(){
  Complex c1(10.0,-10.0),c2(5.0,6.0),c3;
  //Complex c1(10.0,-10.0),c2,c3;//probajte slucaj kada je delilac 0
  c3 = c1+c2; cout<<"("<<c1<<")+("<<c2<<")="<<c3<<endl;
  c3 = c1-c2;  cout<<"("<<c1<<")-("<<c2<<")="<<c3<<endl;
  c3 = c1*c2; cout<<"("<<c1<<")*("<<c2<<")="<<c3<<endl;
  c3 = c1/c2; cout<<"("<<c1<<")/("<<c2<<")="<<c3<<endl;
  cout<<"Konjugovana vrednost "<<c1<<" je "<<(!c1)<<endl;
```

Zadatak

```
string strulaz="ulaz.txt";//ili std::string
ifstream infile (strulaz,ifstream::binary); //ili std::ifstream
if (infile.is_open()) {
  cout<<"sadrzaj fajla "<<strulaz<<" je :"<<endl;
  char c = infile.get();
  while (infile.good()) {
    cout << c;   c = infile.get();
  }
  cout<<endl;
}
else {
  cout<<"Ne moze se otvoriti "<<strulaz<<endl;
  system("pause"); return;
}
string strizlaz="izlaz.txt";
ofstream outfile(strizlaz,ofstream::binary);
if(outfile.is_open()==0){
  cout<<"Ne moze se otvoriti "<<strizlaz<<endl;
  system("pause"); return; }
```

Zadatak

```
  infile.clear();
  infile.seekg (0,ios::end);  //ili infile.end
  long size = infile.tellg();//velicina fajla ulaz.txt
  infile.seekg (0,infile.beg);

  char* buffer = new char[size];//dinamicka alokac. za sadrzaj ulaz.txt
  infile.read (buffer,size);    //ucitavanje sadrzaja ulaz.txt u buffer
  outfile.write (buffer,size);  //snimanje ucitanog sadrzaja u izlaz.txt
  delete[] buffer; //da oslobodite dinamicki blok

  outfile<<c1<<endl;

  outfile.write ("Ovo je kraj    OOPa.",20);
  long pos = outfile.tellp();
  outfile.seekp (pos-13);
  outfile.write ("pocetak",7);

  outfile.close(); infile.close(); system("pause");
}
```

Zadatak

```
_____________________________U_L_A_Z_____________________________________
---> FAJL ulaz.txt
Objektno orijentisano programiranje.



_____________________________I_Z_L_A_Z_____________________________________
(10.00-10.00i)+(5.00+6.00i)=15.00-4.00i
(10.00-10.00i)-(5.00+6.00i)=5.00-16.00i
(10.00-10.00i)*(5.00+6.00i)=110.00+10.00i
(10.00-10.00i)/(5.00+6.00i)=-0.16-1.80i
Konjugovana vrednost 10.00-10.00i je 10.00+10.00i
sadrzaj fajla ulaz.txt je :
Objektno orijentisano programiranje.

---> FAJL izlaz.txt
Objektno orijentisano programiranje.
10.00-10.00iOvo je pocetak OOPa.

Press any key to continue . . .
```

Definisanje izvedene klase u jeziku C++

- Da bi se neka klasa izvela iz neke postojeće klase potrebno je samo da osnovna klasa bude potpuno deklarisana pre deklaracije izvedena klase.
  - Izvedena klasa se deklariše opcionim navođenjem reči public, i obaveznim navođenjem naziva osnovne klase, iza znaka : (dvotačka) u zaglavlju deklaracije klase:

```
class Base {                        // osnovna klasa;
  int i;      public: void f();
};

class Derived : public Base {       // izvedena klasa;
  int j;      public: void g();
};
```

- Objekti izvedene klase nasleđuju sve članove koji čine osnovnu klasu, i još imaju svoje posebne članove koji su navedeni u deklaraciji izvedene klase.

```
void main () {
  Base b;   Derived d;
  b.f();
  b.g();  // ne može; ne postoji g() u klasi Base
  d.f();  // ovo može: d ima i funkciju f,
  d.g(); //                  i funkciju g;
}
```

Definisanje izvedene klase u jeziku C++

- Izvedena klasa jedino ne nasleđuje: konstruktore, destruktore, funkciju članicu operator= i prijatelje osnovne klase.
- Deklaracija člana u izvedenoj klasi sakriva deklaraciju člana sa istim imenom u osnovnoj klasi.
- Klasa A se naziva *direktnom osnovnom klasom* (*direct base class*) neke klase B, ako je ona navedena u listi osnovnih klasa u deklaraciji klase B. Inače, klasa A se naziva *inidrektnom osnovnom klasom*, ako ona nije direktna osnovna klasa klase B, ali je osnovna klasa neke klase koja je navedena u listi osnovnih klasa klase B.
- Notacija *ime_klase*::*ime_člana* se može upotrebiti za pristup nekom članu objekta klase, ako je taj član deklarisan u nekoj od osnovnih klasa.

```
class A { public: void f(); };
class B : public A {};
class C : public B { public: void f(); };

void C::f() {
  f();          // poziv f() iz C;
  A::f();       // poziv f() iz A;
  B::f();       // poziv f() iz A;
}
```

Privatno, zaštićeno i javno izvođenje

- Ključna reč public u zaglavlju deklaracije izvedene klase znači da su javni članovi osnovne klase sada javni članovi izvedene klase a zaštićeni članovi osnovne klase su sada zaštićeni članovi izvedene klase. Funkcije članice izvedene klase ne mogu pristupati privatnim članovima osnovne klase.

```
class Base {
  int pb;
public:
  int jb;
  void put(int x) {pb=x;} };
class Derived: public Base{//Derived je vrsta (a kind of) klase Base.
    int pd;
  public:
    void write(int a, int b, int c) {
      pd=a; jb=b;
      pb=c;    // ovo ne može, jer je pb privatni clan klase Base
      put(c); }// već mora ovako;
};
```

- Zaštićeno izvođenje: javni i zaštićeni članovi osnovne klase postaju zaštićeni članovi izvedene klase. `class B: protected A {/*...*/};`
- Privatno izvođenje (je podrazumevano): javni i zaštićeni članovi osnovne klase postaju privatni članovi izvedene klase. `class B : private A { /*...*/};`

Privatno, zaštićeno i javno izvođenje

- *Zaštićeni članovi* (*protected members*) navode se iza ključne reči protected i oni su dostupni samo izvedenim klasama (i pri sukcesivnom nasleđivanju).
- Ne može se povećati pravo pristupa članu koji je privatan, zaštićen ili javni.

```
class Base {
    int pb;
  protected:
    int zb;
  public:
    int jb;
};
class Derived : public Base { //javno izvodjenje
  //...
public:
  void write(int x) {
    jb=zb=x;  // može da pristupi javnom i zaštićenom članu,
    pb=x;     // ali ne i privatnom: greška!
  }
};
void f() {//globalna funkcija
  Base b;
  b.zb=5; //greška: ne može da se pristupa zaštićenom članu!
}
```

- Javno izvođenje:
  - svi javni članovi osnovne klase su javni članovi izvedene klase.
  - zaštićeni članovi osnovne klase ostaju zaštićeni i u izvedenoj klasi.
  - privatni članovi osnovne klase nisu dostupni izvedenoj klasi,

Prava pristupa

- Pravo pistupa članovima osnovne klase može se promeniti navođenjem imena člana osnovne klase, kvalifikovanog imenom klase (*ime_klase*::*ime_člana*), bez navođenja njegovog tipa, u public ili protected delu izvedene klase.
  - Ova deklaracija ne može niti smanjiti niti povećati nivo prava pristupa koji je član osnovne klase imao u osnovnoj klasi.
- Pravo pristupa nasleđenom članu može se vratiti na nivo koji je imao u osnovnoj klasi.

```
class Base {
  private:            int bpriv;
  protected:          int bprot;
  public:             int bpub;
};
class PrivDerived : private Base {  // privatno izvođenje;
protected:
  Base::bprot; //vraćanje na nivo protected; ne navodi se tip
  Base::bpub;  //greška: smanjenje nivoa prava pristupa!
public:
  Base::bprot; //greška: povećanje nivoa prava pristupa!
};
```

- Klasa **`Base`** je deo (*a-part-of*) klase **`PrivDerived`**.

Izvedene klase i konverzije

- Kaže se da je osnovna klasa *dostupna* (*accessible*) na nekom mestu u programu, ako su na tom mestu dostupni javni članovi te osnovne klase.
- Osnovna klasa A klase B koja je izvedena privatno, dostupna je unutar funkcija članica i prijatelja izvedene klase, ali ne van njih.
- Samo na mestima gde se može pristupati javnom delu osnovne klase, može se pouzdano "znati" da je klasa B izvedena iz klase A.
- Pokazivač na izvedenu klasu se može konvertovati u pokazivač na dostupnu osnovu klasu na mestima na kojima se "zna" da je klasa B izvedena iz klase A.

```
class Base {/*...*/};
class PublicDerived  : public  Base {};
class PrivateDerived : private Base {};
void main () {
  PrivateDerived privd;
  Base *pb1=&privd;  //greška:&privd se ne može konvertovati u Base*!
  PublicDerived pubd;
  Base *pb2=&pubd;  // u redu: osnovna klasa je dostupna;
}
```

- Ako je klasa privatno izvedena, onda se spolja ne može znati da je podobjekat tipa osnovne klase sadržan u objektu izvedene klase, pa zato ovakva konverzija nije dozvoljena.

Izvedene klase i konverzije

- Kada je klasa javno izvedena, onda korisnici spolja mogu znati da je izvedena klasa "jedna vrsta" osnovne klase, jer mogu koristiti sve usluge osnovne klase.

```
class B : public A {...};
```

  - Konverzija iz B* u A* je dozvoljena.
  - Konverzija iz A* u B* nije dozvoljena.

- Ako je B privatno izvedena klasa iz klase A, A se "skriva" unutar B, pa korisnici spolja ne mogu znati da objekat klase B sadrži sve što sadrži i osnovna klasa.

```
class B : private A {...}
```

  - konverzija iz B* u A* nije dozvoljena van funkcija članica i prijatelja klase B

- Potpuno isto pravilo važi i za reference.
- Objekat osnovne klase može se inicijalizovati objektom izvedene klase, ako je osnovna klasa dostupna.

```
class Base {/*...*/};
class Derived : public Base {/*...*/};
void main () {
  Derived d;
  Base b=d;  // u redu: d je istovremeno i tipa Base;
  d=b;       // greška!
}
```

Konstruktori i destruktori izvedenih klasa

- Pri kreiranju objekta izvedene klase, poziva se konstruktor te klase, ali i konstruktor osnovne klase.

```
class Base {
   int bi;
public:
  Base(int);  // konstruktor osnovne klase
};
Base::Base (int i) : bi(i) {/*...*/}
class Derived : public Base {
  int di;
public:
  Derived(int);
};
Derived::Derived (int i) : Base(i), di(i+1) {/*...*/}
```

- Pri kreiranju objekta izvedene klase bira se konstruktor te klase koji će se pozvati. U inicijalizaciji konstruktora, poziva se konstruktor osnovne klase, koji najbolje odgovara stvarnim argumentima u inicijalizatoru.
  - Inicijalizovan je podobjekat osnovne klase unutar objekta izvedene klase.
  - Inicijalizuju se članovi izvedene klase, redosledom kojim su deklarisani, bez obzira na redosled inicijalizatora. Na kraju se izvršava telo konstruktora izvedene klase.

Konstruktori i destruktori izvedenih klasa

- Pri ukidanju objekta izvedene klase, redosled poziva destruktora je obrnut: destruktor izvedene klase, zatim destruktori članova (u redosledu suprotnom od redosleda inicijalizacije) a onda destruktor osnovne klase.

```
#include <iostream>
using namespace std;
class X {
  public:
    X() {cout<<"Konstruktor klase X.\n";}
    ~X(){cout<<"Destruktor klase X.\n";}
};
class Base {
  public:
    Base()  {cout<<"Konstruktor osnovne klase.\n";}
    ~Base() {cout<<"Destruktor osnovne klase.\n";}
};
class Derived : public Base {
    X x;
  public:
    Derived()  {cout<<"Konstruktor izvedene klase.\n";}
    ~Derived() {cout<<"Destruktor izvedene klase.\n";}
};
void main () {  Derived d; d.~Derived(); system("pause"); }
```

```
/* Izlaz će biti:
Konstruktor osnovne klase.
Konstruktor klase X.
Konstruktor izvedene klase.
Destruktor izvedene klase.
Destruktor klase X.
Destruktor osnovne klase.
```

Realizacija izvedenih klasa

```
class A               {  int a;  public:  void f();  };
class B : public A    {  int b;  public:  void g();  };

class C : private B {
  int c;
  public:       void h();
};
```

- Objekti klase C se mogu realizovati na sledeći način: int a; int b; int c;
- Kada se pokazivač na izvedenu klasu, koji sadrži adresu početka strukture podataka u memoriji, konvertuje u pokazivač na objekat osnovne klase, njegova vrednost se ne menja.
- Kada se pokazivač B* konvertuje u A*, rezultujući pokazivač se ne menja.
- Potpuno isto se dešava kada se pokazivač C* konvertuje u pokazivač B*.
  - Pokazivač na objekat tipa C sadrži adresu početka cele strukture podataka, odnosno adresu podobjekta tipa A. Rezultujući pokazivač treba da ukazuje na podobjekat tipa B, koji u sebi sadrži i podobjekat tipa A, koji se opet nalazi na početku cele strukture.
- Redosled smeštanja podobjekata nije specificiran jezikom, pa objekti tipa C mogu izgledati i ovako: int c; int b; int a;

Višestruko izvođenje i višestruki podobjekti

- Ako je neka klasa X višestruka indirektna osnovna klasa klase C, onda će objekti klase C imati višestruke članove klase X, odnosno postojaće više podobjekata klase X unutar jednog objekta klase C.

```
class X {       public: int i;          };

class A : public X {};
class B : public X {};

class C : public A, public B {  public: void f();      };
```

  - U ovom primeru, objekti klase C imaće dva člana sa imenom ***i***.
  - U svakom objektu tipa C postoje dva podobjekta tipa X: jedan je nasleđen od A, a drugi od B.
  - Pristup članu ***i***, bez eksplicitnog navođenja imena klase kojoj pripada, je nedozvoljen, zbog dvosmislenosti.
  - Članu ***i*** može se pristupiti samo eksplicitnim navođenjem imena klase kojoj pripada i operatora razrešavanja opsega važenja:

```
void C::f() {
  i=0;         // greška: dvosmislenost, A::i ili B::i!
  A::i=B::i;   // ovako može;
}
```

Višestruki podobjekti

- Posledica navedenog je da nije dozvoljeno da jedna klasa bude višestruka direktna osnovna klasa, inače se ne bi mogla izbeći dvosmislenost:

```
class C : public B, public B {};  // greška!
```

- Na isti način se tretiraju i konverzije:
  - konverzija pokazivača ili reference na izvedenu klasu u pokazivač ili referencu na osnovnu klasu, dozvoljena je samo ako je osnovna klasa dostupna, i ako je konverzija nedvosmislena.

Za primer sa pethodnog slajda:

```
C *pc=new C;
X *px1=pc;          // greška: dvosmislenost!
X *px2=(X*)pc;      // greška: i dalje je dvosmisleno!
```

- Dvosmislenost se pojavljuje jer se, pri direktnoj konverziji iz C* u X*, ne zna da li rezultujući pokazivač treba da ukazuje na podobjekat tipa X koji pripada podobjektu A ili podobjektu B.
- Rešenje za dvosmislenost:

```
X *px3=(X*)(A*)pc; // u redu: px3 ukazuje na X iz podobjekta A;
X *px4=(X*)(B*)pc; // u redu: px4 ukazuje na X iz podobjekta B;
```

Virtuelne osnovne klase

- Ako je potrebno da izvedena klasa poseduje samo jedan podobjekat indirektne osnovne klase, onda treba tu osnovnu klasu deklarisati kao *virtuelnu* (*virtual base class*):

```
class X {/*...*/};                          //vozilo ima reg.broj
class A : virtual public X {/*...*/};       //motorcikl
class B : virtual public X {/*...*/};       //vozilo sa 3 tocka
class C : public A, public B {/*...*/}; //motorcikl sa prikolicom
```

- U ovom slučaju, objekti klase C imaće samo jedan podobjekat klase X, koga će deliti podobjekti klasa A i B.

```
class Z : public A, public B, public C, public X {/*...*/};
```

  - U ovom primeru, objekat klase Z imaće tri podobjekta tipa X:
    - jedan je zajednički za podobjekte A i B,
    - jedan je odvojeni podobjekat unutar podobjekta C,
    - jedan je sopstveni za objekat Z.

```
class X {/*...*/};
class Y : virtual public X {/*...*/};
class Z : public Y, virtual public X {/*...*/};
```

- Ovde prevodilac učini da objekat klase Z ima samo jedan podobjekat klase X.

Inicijalizacija osnovnih klasa

- Kao i kod jednostrukog izvođenja, višestruke osnovne klase se inicijalizuju navođenjem inicijalizatora u zaglavlju konstruktora izvedene klase.

```
class A {     public: A(int);  };
class B {     public: B(int);  };
class X : public A, public B {
  public:
    X(int i) : A(i+1),B(i+2) {/*...*/}
};
```

- Inicijalizatori osnovnih klasa u zaglavlju konstruktora izvedene klase specificiraju stvarne argumente za pozive konstruktora osnovnih klasa.
  - Kada se kreira objekat izvedene klase, najpre se pozivaju konstruktori osnovnih klasa, po redosledu po kome su deklarisane osnovne klase u zaglavlju deklaracije izvedene klase, bez obzira na redosled inicijalizatora.
  - Zatim se pozivaju konstruktori članova izvedene klase, i na kraju se izvršava samo telo konstruktora izvedene klase. Redosled poziva destruktora je obrnut.
- Navođenjem inicijalizatora u konstruktoru izvedene klase mogu se inicijalizovati samo direktne osnovne klase i članovi koji nisu nasleđeni od osnovnih klasa.

Inicijalizacija osnovnih klasa

```
class V { public: V(int); };
class A : virtual public V {
                        public:    A(int i) : V(i) {/*...*/}
};
class B : virtual public V {
                        public:    B(int i) : V(i) {/*...*/}
};
class X : public A, public B, private virtual V {
  public:
    X(int i) : A(i),B(i),V(i) {/*...*/}
};
```

- U ovom primeru, objekti klase X imaće samo jedan podobjekat virtuelne osnovne klase V.
  - Kada bi osnovne klase A i B, svaka za sebe, inicijalizovale isti podobjekat V koga zajednički poseduju, semantika postojanja samo jednog podobjekta tipa V bila bi narušena. Pošto postoji samo jedan podobjekat virtuelne osnovne klase unutar objekta izvedene klase, potrebno je da se taj podobjekat inicijalizuje samo jedanput.
- Virtuelne osnovne klase inicijalizuju se pre svih nevirtuelnih osnovnih klasa.

Polimorfizam

- Iz osnovne klase figura izvedene su klase krug, kvadrat, trougao itd. Objektima ovih izvedenih klasa pristupa se preko niza pokazivača tipa figura*.
- Funkcija crtanje() izvedene klase *redefiniše* (*overrides*) virtuelnu funkciju (*virtual function*) crtanje() osnovne klase tako da svaka izvedena klasa treba da realizuje funkciju crtanja na sebi svojstven način.

```
void crtanje () {
  for (int i=0; i<broj_figura; i++)
    niz_figura[i]->crtaj();  //niz pokazivaca na objekte figura
}
```

- Funkcija crtanje "proziva" redom elemente niza koje tretira kao objekte osnovne klase figura, iako se radi o posebnoj figuri (krug, trougao itd.).
- Svaka od posebnih vrsta figura iscrtava se na svojstven način, ali to funkcija crtanje uopšte ne mora da "zna". Bitno je da svaka vrsta figure ima svojstvo da se može iscrtati, predstavljeno virtuelnom funkcijom crtaj() osnovne klase.
- Svaki objekat će "prepoznati" kojoj izvedenoj klasi pripada, bez obzira što mu se obraća "uopšteno", kao pokazivaču ili referenci na objekat osnovne klase.
  - Svojstvo da svaki objekat izvedene klase izvršava metod onako kako je to definisano u njegovoj izvedenoj klasi kada mu se pristupa preko pokazivača ili reference na objekat osnovne klase, naziva se *polimorfizam* (*polymorphism*).

Virtuelne funkcije

- Ako se tip funkcije deklarisane u izvedenoj klasi razlikuje od tipa virtuelne funkcije sa istim imenom u osnovnoj klasi, onda funkcija u izvedenoj klasi ne predstavlja novu verziju virtuelne funkcije, već sakriva funkciju osnovne klase. Funkcija izvedene klase, koja **redefiniše** virtuelnu funkciju osnovne klase je isto virtuelna :

```
class B { public:
  virtual void vf1();  virtual void vf2();  virtual void vf3();
  void f();
};
class D : public B { public:
  void vf1();      // nova verzija, redefinisana funkcija;
  void vf2(int);   // sakriva B::vf2; jer void vf2(int) nije virtuelna
  int  vf3();      // greška: razlika u tipu rezultata!
  void f();        // ovo nije virtuelna funkcija;
};
void main() {
  D d;
  B *pb=&d;     // standardna konverzija D* u B*;
  pb->vf1();    // poziva se D::vf1; jer pb ukazuje na D iako je B *pb
  pb->vf2();    // poziva se B::vf2; jer ne postoji void D::vf2()
  pb->f();      // poziva se B::f; jer f nije virtualna
}
```

Virtuelne funkcije

- Poziv virtuelne funkcije se razrešava prema *objektu* na koji ukazuje pokazivač (ili na koga upućuje referenca), a ne prema tipu tog pokazivača (ili reference), dok poziv nevirtuelne funkcije se razrešava u odnosu na tip *pokazivača* ili reference.
- Ako izvedena klasa ne redefiniše svoju verziju virtuelne funkcije, pozivaće se funkcija osnovne klase u svim slučajevima.
- Virtuelna funkcija NE može biti globalna funkcija nečlanica. Takođe, virtuelna funkcija NE može biti statička. Virtuelna funkcija može biti prijatelj druge klase.
- Eksplicitni poziv funkcije sa kvalifikatorom *ime_klase*::*ime_funkcije* ne aktivira virtuelni mehanizam: tada se poziva verzija funkcije iz klase koja je eksplicitno navedena.

```
class B { public:   virtual void f(); };
class D : public B {  //...
  public:
    //...
    void f() {
      B::f();    // eksplicitni poziv osnovne verzije NE aktivira
      //...      // virtualni mehanizma i da se uradi jos nesto
    }
};
```

Dinamičko vezivanje

```
class Base {  //...
public:
  virtual void f();
  //...
};

class Derived : public Base {
  //...
public:
  void f();
};

void g1(Base b)// staticko
  // vezivanje u fazi
  // prevodjenja
{
  b.f();
}

void g2(Base *pb){
  pb->f();
}

void g3(Base &rb){
  rb.f();
}
```

```
void main () {
  Derived d;
  g1(d);          // poziva se Base::f;
  g2(&d);         // poziva se Derived::f;
  g3(d);          // poziva se Derived::f;

  Base *pb=new Derived;//Derived* → Base*
                  // tip objekta = Derived

  pb->f();        // poziva se Derived::f;
                  // prema tipu objekta

  Base &rd=d;     // Derived& → Base&;
                  // tip objekta = Derived
  rd.f();         // poziva se Derived::f;
                  // prema tipu objekta

  Base b=d;       // objekat je Base
  b.f();          // poziva se Base::f;


  delete pb;
  pb=&b;
  pb->f();        // poziva se Base::f;
}
```

U vreme prevođenja se ne zna tačan tip objekta za funkcije g2 i g3, pa se aktivira virtuelni mehanizam (povezivanje je dinamičko) (*dynamic binding*)

Virtuelne funkcije i virtuelne osnovne klase

```
class V {
  public:
    void f();  int x;
};

class B : public virtual V {
  public:
     void f();   int x;
};

class D : public B, public virtual V {
public:
  void g() {
    x++;                        // koji x? od B ili od V
    f();                        // koji f? od B ili od V
  }
};
```

- Razrešavanje obraćanja pravilom dominacije. Iako se V::f i V::x čine podjednako "bliskim" klasi D, imena B::f i B::x dominiraju nad njima, pa se obraćanje x i f odnosi na B::x i B::f.
- Ovakvi slučajevi rešavaju se tzv. pravilom dominacije. Kaže se da neko ime D::f, u opštem slučaju, dominira nad imenom B::f, ako je B osnovna klasa klase D. U prethodnom primeru, imena B::f i B::x dominiraju nad imenima V::f i V::x.

Pravila vezana za virtuelne funkcije

- Kontrola prava pristupa virtuelnoj funkciji vrši se u skladu sa njenom (prvom) deklaracijom, i na ta prava ne utiču pravila vezana za funkcije koje je kasnije redefinišu. Na primer:

```
class B {
public:
  virtual void f();  // virtuelna funkcija f;
};

class D : public B {
private:
  void f();          // redefinisana funkcija f; i ona je virtuelna
};

void main () {
  D d;
  B *pb=&d;
  pb->f();   // u redu: B::f je javna, a poziva se D::f;
  D *pd=&d;
  pd->f();   // greška: D::f je privatna!
}
```

- U primeru, za pristup preko pokazivača pb (pokazivača B*), pristup javnoj funkciji f je dozvoljen, iako se virtuelnim mehanizmom pristupa funkciji D::f.
- U drugom slučaju, tip pokazivača je D*, pa pristup privatnoj funkciji f nije dozvoljen.

Pravila vezana za virtuelne funkcije

- Unutar konstruktora i destruktora mogu se pozivati funkcije članice, pa i virtuelne funkcije.

```
class B {
public:
  virtual void f();
  B()  { f(); }        // iz konstruktora poziva se uvek B::f;
  ~B() { f(); }        // iz destruktora  poziva se uvek B::f;
};
class D : public B {
  int &r;
public:
   D(int &rr) : r(rr) {} //poziv konstruktora za B, inicij. reference
   void f() { r++; }     //redefinisana funkcija D::f
};
void main () {    int i=0;     D d(i); }
```

- Redefinisana funkcija D::f se oslanja na činjenicu da je objekat klase D, kome ona pripada, propisno i do kraja konstruisan.
- Kada se kreira objekat d klase D, poziva se najpre konstruktor klase B koji inicijalizuje podobjekat tipa B unutar objekta d.
- Kada bi se u konstruktoru klase B pozivala verzija D::f, nastala bi greška, jer je u tom trenutku deo objekta d koji nije nasleđen još uvek neinicijalizovan.