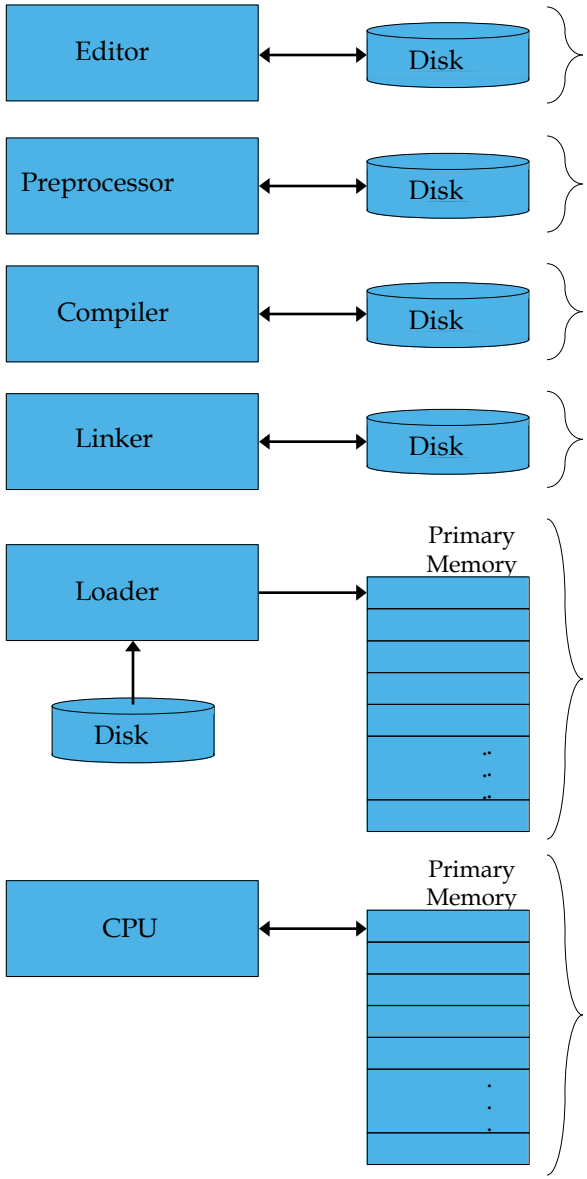
Objektno Orijentisano Programiranje - C++

Prof. Dragan Janković

Elementi standardnog C++ okruženja

Faze u razvoju C++ programa:

1. **Editovanje**
2. **Preprocesiranje**
3. **Compajliranje**
4. **Linkovanje**
5. **Učitavanje**
6. **Izvršavanje**

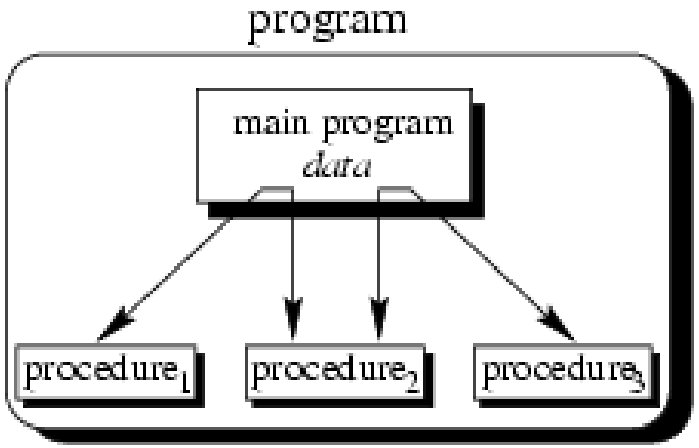


Pregled tehnika programiranja

- **Nestrukturno - proceduralno programiranje**
  - **Glavni program direktno operiše sa globalnim podacima.**
  - **Dugi i nepregledni programi**
  - **Copy paste- Kod se višestruko koristi kopiranjem delova**

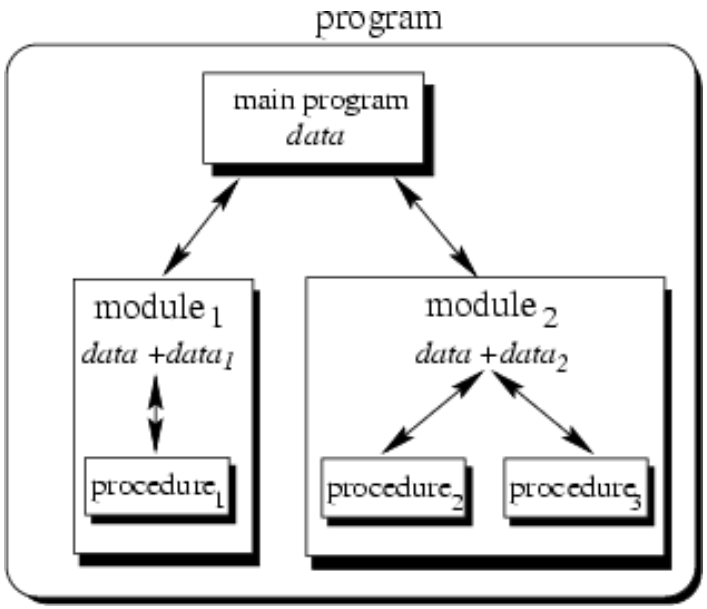
Pregled tehnika programiranja

- **Proceduralno programiranje**
  - **Program se može posmatrati kao sekvenca poziva potprograma (procedura).**
  - **Strukture podataka se modeliraju odvojeno od koda procedura koje ih obrađuju.**
  - **Višestruko korišćenje koda postiže se preko biblioteka procedura i funkcija.**



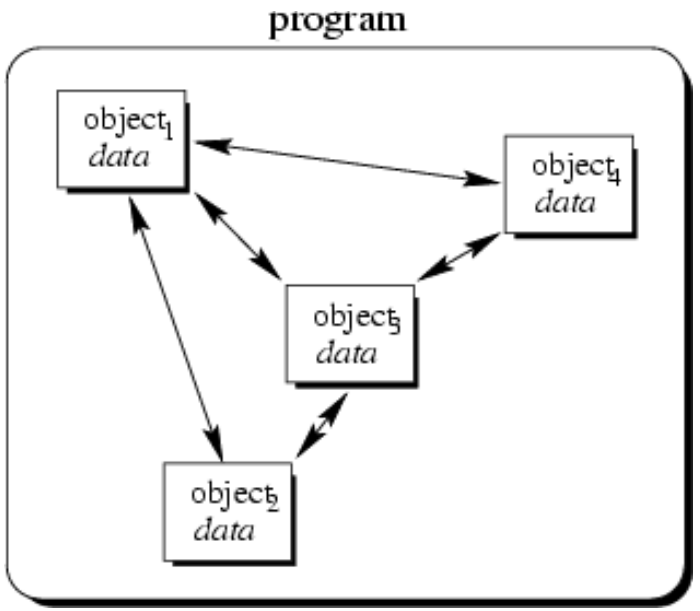
Pregled tehnika programiranja

- **Modularno programiranje**
  - **Procedure sa zajedničkom funkcionalnošću su integrisane u jedan modul**
  - **Svaki modul može da ima svoje sopstvene podatke.**
  - **Višestruko korišćenje struktura podataka i procedura**



Pregled tehnika programiranja

- **Objektno orijentisano programiranje**
  - **Strukture podataka i procedure integrisane u klase**
  - **Program može da se posmatra kao mreža objekata koji su u interakciji pri čemu svaki objekat zna svoje stanje.**
  - **Apstracija, enkapsulacija, nasleđivanje i polimorfizam**
  - **Ponovno korišćenje objekata**



Modelovanje problema

- Definisati objekte koji se sreću u opisu problema
- Apstrahovati objekte klasama
- Definisati elemente klasa
- Definisati broj objekata, trenutak njihovog nastajanja, nestajanja i način medjusobne interakcije tokom vremena
- Definisati odgovornosti

Modelovanje problema klasama

- Klase
  - Modeluju (apstrahuju) objekte
    - Atribute (podaci članovi)
    - Ponašanja (funkcije članice)
  - Definišu se upotrebom ključne reči class
  - Funkcije članice
    - Metode
    - Aktiviraju se kao odgovori na poruke
- Specifikatori prava pristupa članovima klase
  - public:
    - Pristup omogućen svuda gde objekat klase ima scope
  - private:
    - Pristup omogućen samo u funkcijama članicama
  - protected:
    - U izvedenim klasama

- **Konstruktori**
  - **Specijalne funkcije članice**
    - **Inicijalizuju podatke članove klase**
    - **Isto ime kao i ime klase**
  - **Pozivaju se kada se kreira (instancira) objekat klase**
  - **Više konstruktora**
    - **Overloading funkcija**
  - **Nemaju povratni tip**

```
1     class Time {
2
3     public:
4        Time();                          // constructor
5        void setTime( int, int, int ); // set hour,
   minute, second
6        void printUniversal();          // print
   universal-time format
7        void printStandard()                  t standard-
   time format
8
9     private:
10       int hour;     // 0 - 23 (24-hour clock format)
11       int minute;   // 0 - 59
12       int second;
13
14    }; // end class Time
```

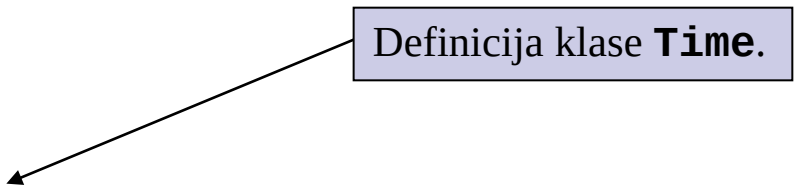
tupa.

i klasa,

- **Funkcije članice definisane van klase**
  - **Binarni rezolucioni operator (::)**
    - **"Vezuje" ime funkcije sa imenom klase**
    - **Jedinstveno identifikuje funkcije konkretne klase**
    - **Različite klase mogu imati funkcije članice sa istim imenima**
  - **Format za definisanje funkcija članica**

    ```
    PovratniTip ImeKlase::ImeFunkcijeČlanice( ){
        ...
    }
    ```
  - **Ne zavisi da li je funkcija public ili private**
- **Funkcije članice definisane unutar klase**
  - **Nije potrebno navoditi operator :: niti ime klase**

```
1
2     // klasa Time.
3     #include <iostream>
4
5     using std::cout;
6     using std::endl;
7
8     #include <iomanip>
9
10    using std::setfill;
11    using std::setw;
12
13    // Definicija apstraktnog tipa Time
14    class Time {
15
16    public:
17       Time();                          // konstruktor
18       void setTime( int, int, int ); // set hour, minute, second
19       void printUniversal();           // print universal-time format
20       void printStandard();            // print standard-time format
21
```



```
22    private:
23       int hour;     // 0 - 23 (24-hour clock format)
24       int minute;   // 0 - 59
25       int second;   // 0 - 59
26
27    }; // end class Time
28
29    // Konstruktor Time inicijalizuje sve podatke članove na nulu
30    // obezbeđujući da svi objekti klase Time imaju isti oblik
31    Time::Time()
32    {
33       hour = minute = second = 0;
34
35    } // end Time constructor
36
37    // set new Time value using universal time, p
38    // checks on the data values and set invalid
39    void Time::setTime( int h, int m, int s )
40    {
41       hour = ( h >= 0 && h < 24 ) ? h : 0;
42       minute = ( m >= 0 && m < 60 ) ? m : 0;
43       second = ( s >= 0 && s < 60 ) ? s : 0;
44    } // end function setTime
45
```

Konstruktor inicijalizuje **`private`** podatak na **`0`**.

`public` funkcija članica proverava vrednost parametara pre setovanja **`private`** podataka.

```
47    // print Time in universal format
48    void Time::printUniversal()
49    {
50       cout << setfill( '0' ) << setw( 2 ) << hour << ":"
51            << setw( 2 ) << minute << ":"
52            << setw( 2 ) << second;
53
54    } // end function printUniversal
55
56    // print Time in standard format
57    void Time::printStandard()
58    {
59       cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
60            << ":" << setfill( '0' ) << setw( 2 ) << minute
61            << ":" << setw( 2 ) << second
62            << ( hour < 12 ? " AM" : " PM" );
63
64    } // end function printStandard
65
66    int main()
67    {
68       Time t;  // instanciranje (kreiranje) objekta t klase Time
69
```

Nema argumenata.

Deklaracija promenljive **t** kao objekta klase **Time**.

```
70      // output Time object t's initial values
71      cout << "The initial universal time is ";
72      t.printUniversal();   // 00:00:00
73
74      cout << "\nThe initial standard time is ";
75      t.printStandard();    // 12:00:00 AM
76
77      t.setTime( 13, 27, 6 );   // change time
78
79      // output Time object
80      cout << "\n\nUniversal
81      t.printUniversal();
82
83      cout << "\nStandard time after setTi
84      t.printStandard();    // 1:27:06 PM
85
86      t.setTime( 99, 99, 99 );  // attempt
87
88      // output t's values after specifying invalid values
89      cout << "\n\nAfter attempting invalid settings:"
90           << "\nUniversal time: ";
91      t.printUniversal();   // 00:00:00
92
```

poziv **public** metode

Postavljanje vrednosti člana klase korišćenjem **public** funckije članice.

Pokušaj postavljanja vrednosti člana klase na nevalidnu vrednost korišćenjem **public** funkcije članice.

```
93        cout << "\nStandard time: ";
94        t.printStandard();    // 12:00:00 AM
95        cout << endl;
96
97        return 0;
98
```

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

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

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Podatak postavljen na **0** nakon pokušaja postavljanja nevalidne vrednosti.

Inline Funkcije

- Inline funkcije
  - Ključna reč **inline** ispred funkcije
  - Nalog kompajleru da kopira kod u program umesto da se generiše poziv funkcije
    - Umanjuje broj poziva funkcija
    - Kompajler može da ignoriše **inline**
  - Dobro za male funkcije koje se često koriste
- **Primer:**

```
inline double cube( const double s )
    { return s * s * s; }
```

  - **const** kazuje kompajleru da funkcija ne menja **s**

```
2    // Using an inline function to calculate.
3    // the volume of a cube.
4    #include <iostream>
6    using std::cout;
7    using std::cin;
8    using std::endl;
10   // Definition of inline function cube. Definition of function
11   // appears before function is called, so a function prototype
12   // is not required. First line of function definition acts as
13   // the prototype.
14   inline double cube( const double side )
15   {
16      return side * side * side;  // calculate cube
18   } // end function cube
20   int main()
21   {
22      cout << "Enter the side length of your cube: ";
24      double sideValue;
26      cin >> sideValue;
28      // calculate cube of sideValue and display result
29      cout << "Volume of cube with side "
30           << sideValue << " is " << cube( sideValue ) << endl;
31
32      return 0;  // indicates successful termination
33
34   } // end main
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

- **Destruktori**
  - Isto ime kao i klasa
    - Imenu prethodi simbol tilda (~)
  - Nemaju argumente
  - Ne mogu biti overloadovani (preklopljeni)
  - Izvršavaju radnje kod uništavanja objekata
- **Prednosti korišćenja klasa**
  - Jednostavnije programiranje
  - Interfejs
    - Skrivanje implementacije
  - Ponovno korišćenje softvera
    - Kompozicija (agregacija)
      - Objekti klase mogu da uključuju objekte drugih klasa
    - Izvodjenje (nasledjivanje)
      - Nove klase se izvode iz postojećih

Class Scope i pristup članovima klase

- Class scope
  - Podaci članovi, funkcije članice
  - U okviru class scope
    - Članovi klase
      - Direktno dostupni svim funkcijama članicama
      - Referenciranje navodjenjem imena
  - Van class scope
    - Referenciranje na osnovu
      - Imena objekta, reference objekta, pointera na objekat
- File scope
  - Funkcije koje nisu članice

- **Scope funkcije**
  - **Promenljive deklarisane u funkciji članici**
  - **Poznate samo u funkciji**
  - **Promenljive sa istim imenom kao promenljive u class-scope**
    - **Class-scope promenljive se "sakrivaju - maskiraju"**
      - **Pristup je moguć korišćenjem scope resolution operatora (::)**

        ImeKlase::imePromenljiveKlase
  - **Promenljive su poznate samo u funkcijama gde su definisane**
  - **Promenljive se uništavaju nakon završetka funkcije tj. izlaska iz nje**

- Operatori za pristup članovima klase
  - Identični onima kod struktura (struct)
  - Operator tačka (.)
    - Objekat
    - Referenca objekta
  - Operator strelica (->)
    - Pointeri

Razdvajanje Interfejsa od Implementacije

- Razdvajanje interfejsa od implementacije
  - Prednosti
    - Lakše se modifikuju programi
  - Nedostaci
    - Header fajlovi
      - Delovi implementacije
        - Inline funkcije članice
      - Napomene o ostalim implementacijama
        - private članovi
    - Viši stepen skrivanja korišćenjem proxy klase

- Header fajlovi
  - Definicija klasa i prototipova funkcija
  - Uključuju se u svaki fajl koji koristi klasu
    - #include
  - Ekstenzija fajla .h
- Fajlovi sa kodom
  - Definicije funkcija članica
  - Isto osnovno ime
    - konvencija
  - Kompajliraju se i posle se linkuju

```
1
2     // Declaration of class Time.
3     // Member functions are d
4
5     // prevent multiple inclu
6     #ifndef TIME1_H
7     #define TIME1_H
8
9     // Time abstr
10    class Time {
11
12    public:
13       Time();                             // constructor
14       void setTime( int, int, int );      // set hour, minute, second
15       void printUniversal();              // print universal-time format
16       void printStandard();               // print standard-time format
17
18    private:
19       int hour;     // 0 - 23 (24-hour clock format)
20       int minute;   // 0 - 59
21       int second;   // 0 - 59
22
23    }; // end class Time
24
25    #endif
```

Pretprocesorska direktiva kao zaštita od višestukog uključivanja

"If not d

Preproce
definiše

konvencija:
ime header fajla sa underscore.

```
1
2  // Member-function definitions for class Time.
3  #include <iostream>
4
5  using std::cout;
6
7  #include <iomanip>
8
9  using std::setfill;
10 using std::setw;
11
12 // include definition of class Time from time1.h
13 #include "time1.h"
14
15 // Time constructor initializes ea
16 // Ensures all Time objects start i
17 Time::Time()
18 {
19    hour = minute = second = 0;
20
21 } // end Time constructor
22
```

Uključivanje header fajla **`time1.h`**.

Ime header fajla u duplim navodnicima; uglasti navodnici se koriste za header fajlove iz C++ Standard Library.

```
23    // Set new Time value using universal time. Perform validity
24    // checks on the data values. Set invalid values to zero.
25    void Time::setTime( int h, int m, int s )
26    {
27       hour = ( h >= 0 && h < 24 ) ? h : 0;
28       minute = ( m >= 0 && m < 60 ) ? m : 0;
29       second = ( s >= 0 && s < 60 ) ? s : 0;
30
31    } // end function setTime
32
33    // print Time in universal format
34    void Time::printUniversal()
35    {
36       cout << setfill( '0' ) << setw( 2 ) << hour << ":"
37            << setw( 2 ) << minute << ":"
38            << setw( 2 ) << second;
39
40    } // end function printUniversal
41
```

```
42    // print Time in standard format
43    void Time::printStandard()
44    {
45       cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
46            << ":" << setfill( '0' ) << setw( 2 ) << minute
47            << ":" << setw( 2 ) << second
48            << ( hour < 12 ? " AM" : " PM" );
49
50    } // end function printStandard
```

Poziv Konstruktora i Destruktora

- Konstruktori i destruktori
  - Implicitno se pozivaju - kompajler
- Redosled poziva
  - Zavisi od redosleda izvršavanja
    - Kada u toku izvršavanja objekti dobijaju odnosno gube scope (oblast važenja, doseg)
  - Generalno, destruktori se pozivaju u suprotnom redosledu od redosleda poziva konstruktora

Poziv Konstruktora i Destruktora

- Redosled poziva konstruktora i destruktora
  - Globalni objekti
    - Konstruktori
      - Pre bilo koje druge funkcije (uključujući i main)
    - Destruktori
      - Kada se main završava (ili je pozvana exit funkcija)
      - Ne poziva se ako se program završi sa abort
  - Automatski lokalni objekti
    - Konstruktori
      - Kada se objekti definišu
        - Svaki put kada u toku izvršenja dobiju scope
    - Destruktori
      - Kada objekti prestanu da važe tj gube scope
        - Kada se završava blok u kome je objekat definisan
      - Ne poziva se ako se program završi sa exit ili abort

- Statički lokalni objekti
  - Konstruktori
    - Tačno jedanput
    - Kada se dodje do mesta gde je objekat definisan
  - Destruktori
    - Kada se main završava ili je pozvana exit funkcija
    - Ne poziva se ako se program završava sa abort

```
1
2       // Definition of class CreateAndDestroy.
3       // Member functions defined in create.cpp.
4       #ifndef CREATE_H
5       #define CREATE_H
6
7       class CreateAndDestroy {
8
9       public:
10        CreateAndDestroy( int, char * );  // constructor
11        ~CreateAndDestroy();              // destructor
12
13    private:
14       int objectID;
15       char *message;
16
17    }; // end class CreateAndDestroy
18
19    #endif
```

ili #pragma once

```
1
2       // Member-function definitions for class CreateAndDestroy
3       #include <iostream>
4
5       using std::cout;
6       using std::endl;
7
8       // include CreateAndDestroy class definition from create.h
9       #include "create.h"
10
11    // constructor
12    CreateAndDestroy::CreateAndDestroy(
13       int objectNumber, char *messagePtr )
14    {
15       objectID = objectNumber;
16       message = messagePtr;
17
18       cout << "Object " << objectID << "  constructor runs   "
19            << message << endl;
20
21    } // end CreateAndDestroy constructor
22
```

```
23    // destructor
24    CreateAndDestroy::~CreateAndDestroy()
25    {
26
27       cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );
28
29       cout << "Object " << objectID << "  destructor runs    "
30            << message << endl;
31
32    } // end ~CreateAndDestroy destructor
```

```
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>

using std::cout;
using std::endl;

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

void create( void );   // prototype

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

int main()
{
   cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;

   CreateAndDestroy second( 2,"(local automa

   static CreateAndDestroy third(3,"(local static in main)" );
```

Kreiranje globalnog objekta.

Automatsi lokalni objekat

Kreiranje **static** lokalnog objekta.

```
   create();  // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;

   CreateAndDestroy fourth( 4, "(local automatic in main)" );

   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;

   return 0;
} // end main

// function to create objects
void create( void )
{
   cout << "\nCREATE FUNCTION: EXECUTION

   CreateAndDestroy fifth( 5, "(local au                    );

   static CreateAndDestroy sixth(6, "(lo                    )" );

   CreateAndDestroy seventh(7, "(local automatic in create)" );

   cout << "\nCREATE FUNCTION: EXECUTION ENDS\" << endl;
} // end function create
```

Kreiranje lokalnih automatskih objekata.

Kreiranje lokalnog automatskog objekta.

Kreiranje lokalnog automatskog objekta u

Kreiranje **static** lokalnog objekta u funkciji.

Kreiranje lokalnog automatskog objekta u funkciji.

```
Object 1   constructor runs   (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs   (local automatic in main)
Object 3   constructor runs   (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5   constructor runs   (local automatic in create)
Object 6   constructor runs   (local static in create)
Object 7   constructor runs   (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7   destructor runs    (local automatic in create)
Object 5   destructor runs    (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs    (local automatic in main)
Object 2   destructor runs    (local automatic in main)
Object 6   destructor runs    (local static in create)
Object 3   destructor runs    (local static in main)

Object 1   destructor runs    (global before main)
```

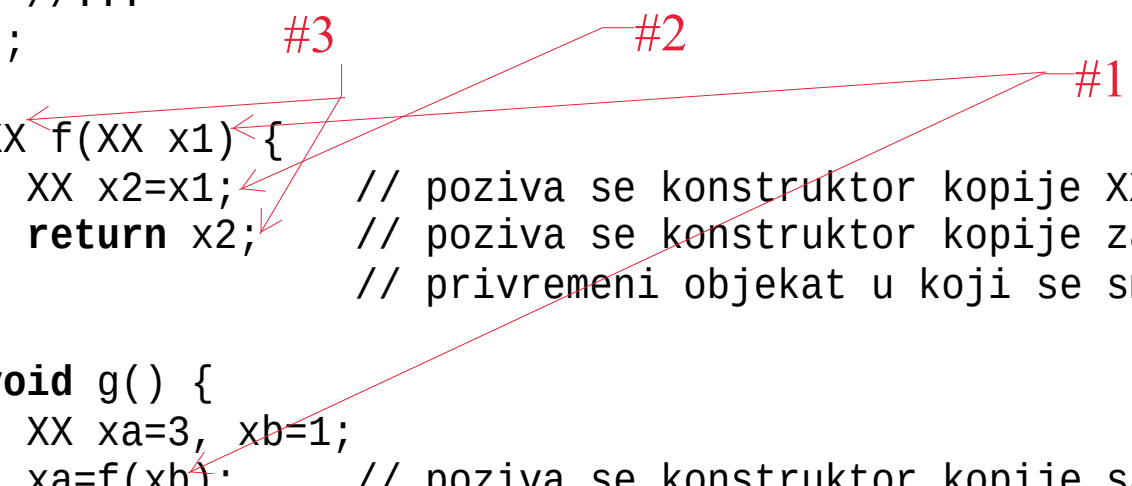
Podrazumevana pokomponentna dodela

- int x,z; x=z; KlasaC x, z; x=z; ????
- Dodela objekata
  - Operator dodele (=)
    - Objekat se može dodeliti objektu istog tipa
    - Default: pokomponentna dodela
      - Svaki desni član se pojedinačno dodeljuje levom
- Prosledjivanje, vraćanje objekata
  - Objekti kao argumenti funkcija
  - Objekti kao vraćene vrednosti iz funkcija
  - Default: prosledjivanje po vrednosti (pass-by-value)
    - Kopija objekta se prosledjuje, vraća
      - Copy konstruktor
        - Kopira originalne vrednosti u novi objekat

Konstruktor kopije

- Kada se objekat `x1` klase `XX` inicijalizuje drugim objektom `x2` iste klase, C++ će podrazumevano izvršiti prostu inicijalizaciju redom članova objekta `x1` članovima objekta `x2`.
- To ponekad nije zadovoljavajuće (često ako objekti sadrže članove koji su pokazivači ili reference),
  pa programer treba da ima potpunu kontrolu nad inicijalizacijom objekta drugim objektom iste klase.
- Za ovu svrhu služi tzv. konstruktor kopije (engl. *copy constructor*).
- Konstruktor kopije klase `XX` se može pozvati sa samo jednim stvarnim argumentom tipa `XX`.
- Konstruktor kopije se poziva kada se objekat inicijalizuje objektom iste klase, a to je:
  1. prilikom inicijalizacije objekta (pomoću znaka = ili sa zagradama);
  2. prilikom prenosa argumenata u funkciju (kreira se lokalni automatski objekat);
  3. prilikom vraćanja vrednosti iz funkcije (kreira se privremeni objekat).
- Konstruktor ne sme imati formalni argument tipa `XX`.
- Konstruktor kopije ima argument tipa `XX&` ili (najčešće) `const XX&`.
- Ostali eventualni argumenti kopirajućeg konstruktora moraju imati podrazumevane vrednosti.

```
class XX {
public:
  XX (int);
  XX (const XX&);     // konstruktor kopije
  //...
};

XX f(XX x1) {
  XX x2=x1;       // poziva se konstruktor kopije XX(XX&) za x2
  return x2;      // poziva se konstruktor kopije za
}                 // privremeni objekat u koji se smešta rezultat

void g() {
  XX xa=3, xb=1;
  xa=f(xb);       // poziva se konstruktor kopije samo za formalni argument x1,
                  // a u xa se samo prepisuje privremeni objekat rezultata,
}                 // ili se poziva XX::operator= ako je definisan
```



Prijatelji klasa

- Ponekad je potrebno da klasa ima i "povlašćene" korisnike koji mogu da pristupaju njenim privatnim članovima.
- Povlašćeni korisnici su funkcije (globalne ili članice drugih klasa) ili cele klase.
- Takve funkcije i klase nazivaju se *prijateljima* (engl. *friends*).
- Prijateljstvo se **ne nasleđuje**, **nije simetrična** i **nije tranzitivna relacija**.

Prijateljske funkcije

- Prijateljske funkcije su funkcije koje nisu članice klase, ali imaju pristup do privatnih članova klase.
- Prijateljske funkcije mogu da budu:
  - globalne funkcije ili
  - članice drugih klasa.
- Funkcija je prijateljska ako se u definiciji klase navede deklaracija funkcije sa ključnom reči **friend** ispred.
- Nevažno je da li se deklaracija prijateljske funkcije navodi u privatnom ili javnom delu klase.

- Prijateljska funkcija nema pokazivač this na objekat klase kojoj je prijatelj.

- Prijateljstvo je relacija koja reguliše pravo pristupa, a ne oblast važenja i vidljivost identifikatora.

- Funkcija može da bude prijatelj većem broju klasa istovremeno.

- U nekim situacijama su globalne prijateljske funkcije pogodnije od funkcija članica:
  1. funkcija članica mora da se pozove za objekat date klase, dok globalnoj funkciji može da se dostavi i objekat drugog tipa, koji će se konvertovati u potrebni tip;
  2. kada funkcija treba da pristupa članovima više klasa, efikasnija je prijateljska globalna funkcija;
  3. ponekad je notaciono pogodnije da se koriste globalne funkcije **(f(x))** nego članice **(x.f())**;
     na primer: **max(a,b)** je čitljivije od **a.max(b)**;
  4. kada se preklapaju operatori, često je jednostavnije definisati globalne (operatorske) funkcije nego članice.

```
class X {
  friend void g (int X&);  // prijateljska globalna funkcija
  friend void Y::h ();     // prijateljska clanica h druge klase Y
  int i;
public:
  void f(int ip) {i=ip;}
};

void g (int k, X &x) {
  x.i=k;             // prijateljska funkcija moze da pristupa
}                    // privatnim clanovima klase

void main () {
  X x;
  x.f(5);            // postavljanje preko clanice
  g(6,x);            // postavljanje preko prijatelja
}
```

Prijateljske klase

- Ako su sve funkcije članice klase `Y` prijateljske funkcije klasi `X`, onda je `Y` prijateljska klasa (*friend class*) klasi `X`.

```
class X {
  friend Y;        // ako je klasa Y definisana ili deklarisana
  friend class Y; // ako klasa Y jos nije ni definisana ni deklarisana
};
```

- Sve funkcije članice klase `Y` mogu da pristupaju privatnim članovima klase `X`.
- Prijateljske klase se tipično koriste kada neke dve klase imaju tešnje međusobne veze.
- Na primer, na sledeći način može se obezbediti da samo klasa `Kreator` može da kreira objekte klase `X`:

```
class X {
 private:
  friend class Kreator;
  X();  // konstruktor je dostupan samo klasi Kreator
};
```

Preklapanje operatora

Pojam preklapanja operatora

- Pretpostavimo da su u programu potrebni kompleksni brojevi i operacije nad njima.
- Tip kompleksnog broja će realizovati klasa koja sadrži elemente (real, imag), a takođe i funkcije za operacije.
- Pogodno je da se pozivi funkcija koje realizuju operacije mogu notacijski predstaviti standardnim operatorima.
- U jeziku C++, operatori za korisničke tipove (klase) su specijalne operatorske funkcije.
- Operatorske funkcije nose ime `operator@`, gde je `@` neki operator ugrađen u jezik.
- Operatorske funkcije preklapaju standaradne operatore (+, -, *, /, ...).

```
class complex {
public:
  complex(double,double);                              /* konstruktor */
  friend complex operator+(complex,complex);           /* oparator + */
  friend complex operator-(complex,complex);           /* operator - */
private:
  double real, imag;
};

complex::complex (double r, double i) : real(r), imag(i) {}

complex operator+ (complex c1, complex c2) {
  complex temp(0,0);                 // privremena promenljiva tipa complex
  temp.real=c1.real+c2.real;
  temp.imag=c1.imag+c2.imag;
  return temp;
}

complex operator- (complex c1, complex c2) {
  return complex(c1.real-c2.real,c1.imag-c2.imag); //poziv konstruktora
}
```

- Operatorske funkcije se mogu koristiti u izrazima kao i operatori nad ugrađenim tipovima.
- Ako je operatorska funkcija definisana na gornji način izraz `t1@t2` se tumači kao `operator@(t1,t2)`:

```
complex c1(3,5.4),c2(0,-5.4),c3(0,0);
c3=c1+c2;      /* poziva se operator+(c1,c2) */
c1=c2-c3;      /* poziva se operator-(c2,c3) */
```

Operatorske funkcije

Osnovna pravila

- C++ dozvoljava preklapanje operatora (*operator overloading*), kao što dozvoljava i preklapanje imena funkcija.
- Princip preklapanja omogućava da se definišu nova značenja operatora za korisničke tipove.
- Postoje neka ograničenja u preklapanju operatora:
  1. ne mogu da se preklope operatori `.`, `.*`, `::`, `?:` i `sizeof`, dok svi ostali mogu;
  2. ne mogu da se redefinišu značenja operatora za primitivne (standardne) tipove podataka;
  3. ne mogu da se uvode novi simboli za operatore;
  4. ne mogu da se menjaju osobine operatora koje su ugrađene u jezik: *n*-arnost, prioritet i asocijativnost.

```
complex operator+ (complex c, double d) {   // ispravno
  return complex(c.real+d,c.imag);
}
 complex operator** (complex c, double d) { // ! GRESKA: ne postoji stepenovanje
}
```

- Operatorske funkcije mogu biti:
  1. funkcije članice kod kojih je skrivreni argument levi operand ili
  2. globalne funkcije (uglavnom prijatelji klasa) kod kojih je bar jedan argument tipa korisničke klase.
- Za korisničke tipove su unapred definisana tri operatora: = (dodela vrednosti), & (uzimanje adrese) i , (lančanje).
  - Sve dok ih korisnik ne redefiniše, oni imaju podrazumevano značenje.
- Podrazumevano značenje operatora = je kopiranje objekta član po član.
  - Pri kopiranju članova tipa klase, pozivaju se operatori = klasa kojima članovi pripadaju.
  - Ako je član objekta pokazivač, kopiraće se samo taj pokazivač, a ne i pokazivana vrednost.
  - Kada treba kopirati i objekat na koji ukazuje član-pokazivač korisnik tada treba da redefiniše operator =.
- Vrednosti operatorskih funkcija mogu da budu bilo kog tipa, pa i `void`.
- Ako se simbol operatora sastoji od slova (npr `new`), mora se pisati odvojeno od ključne reči `operator`.

Bočni efekti i veze između operatora

- Bočni efekti koji postoje kod operatora za ugrađene tipove nikad se ne podrazumevaju za redefinisane operatore.
- To važi za operatore ++ i `--` (prefiksne i postfiksne) i sve operatore dodele (=, +=, -=, *= itd.).
- Operator = (i ostali operatori dodele) ne mora da menja stanje objekta; ipak, ovakve upotrebe treba izbegavati.
- Strogo se preporučuje da operatori koje definiše korisnik imaju očekivano značenje, radi čitljivosti programa.
  - Na primer, ako su definisani i operator += i operator +, dobro je da `a+=b` ima isti efekat kao i `a=a+b`.
- Veze koje postoje između operatora za ugrađene tipove se ne podrazumevaju za redefinisane operatore.
  - Na primer, ako je definisan operator +, `a+=b` ne znači automatski `a=a+b`, (+= mora posebno da se definiše).
- Kada se definišu operatori za klasu, treba težiti da njihov skup bude kompletan.
  - Na primer, ako su definisani operatori = i +, treba definisati i operator +=; ili, za == treba definisati i `!=`.

Operatorske funkcije kao članice ili kao globalne funkcije

- Ako je @ neki binarni operator (na primer +), on može da se realizuje:
  1. Kao funkcija članica klase `X` (mogu se argumenti prenositi i po referenci): `<tip> operator@ (X)`
     Poziv `a@b` se sada tumači kao: `a.operator@(b)`, za funkciju članicu, ili
  2. Kao prijateljska globalna funkcija:
     `<tip> operator@ (X,X)`
     Poziv `a@b` se sada tumači kao: `operator@(a,b)`, za globalnu funkciju
- Nije dozvoljeno da se u programu nalaze obe ove funkcije.

```
class complex {
  double real,imag;
public:
  complex (double r=0, double i=0) : real(r), imag(i) {}
  complex operator+(complex c) { return complex(real+c.real,imag+c.imag; }
};

// ili, alternativno:
class complex {
  double real,imag;
public:
  complex (double r=0, double i=0) : real(r), imag(i) {}
  friend complex operator+(complex,complex);
};
complex operator+ (complex c1, complex c2) {
  return complex(c1.real+c2.real,c1.imag+c2.imag);
}

void main () {
  complex c1(2,3),c2(3.4);
  complex c3=c1+c2;  // poziva se c1.operator+(c2) ili operator+(c1,c2)
}
```

- Kod operatorske funkcije članice klase levi operand je skriveni argument – objekat date klase.
- Ako levi operand treba da bude standardnog tipa mora se definisati prijateljska funkcija u klasi drugog argumenta.
  - Na primer: `complex operator-(double d, complex c)` mora biti prijateljska, a ne članica.
- Operatorska funkcija članica ne dozvoljava konverziju levog operanda.

Unarni i binarni operatori

- Unarni operator ima samo jedan operand, pa se može realizovati:
  1. kao operatorska funkcija članica bez argumenata:
     ```
     tip operator@ ()
     ```
  2. kao globalna funkcija sa jednim argumentom:
     ```
     tip operator@ (X x)
     ```
- Binarni operator ima dva argumenta, pa se može realizovati
  1. kao funkcija članica sa jednim argumentom:
     ```
     tip operator@ (X xdesni)
     ```
  2. kao globalna funkcija sa dva argumenta:
     ```
     tip operator@ (X xlevi, X xdesni)
     ```

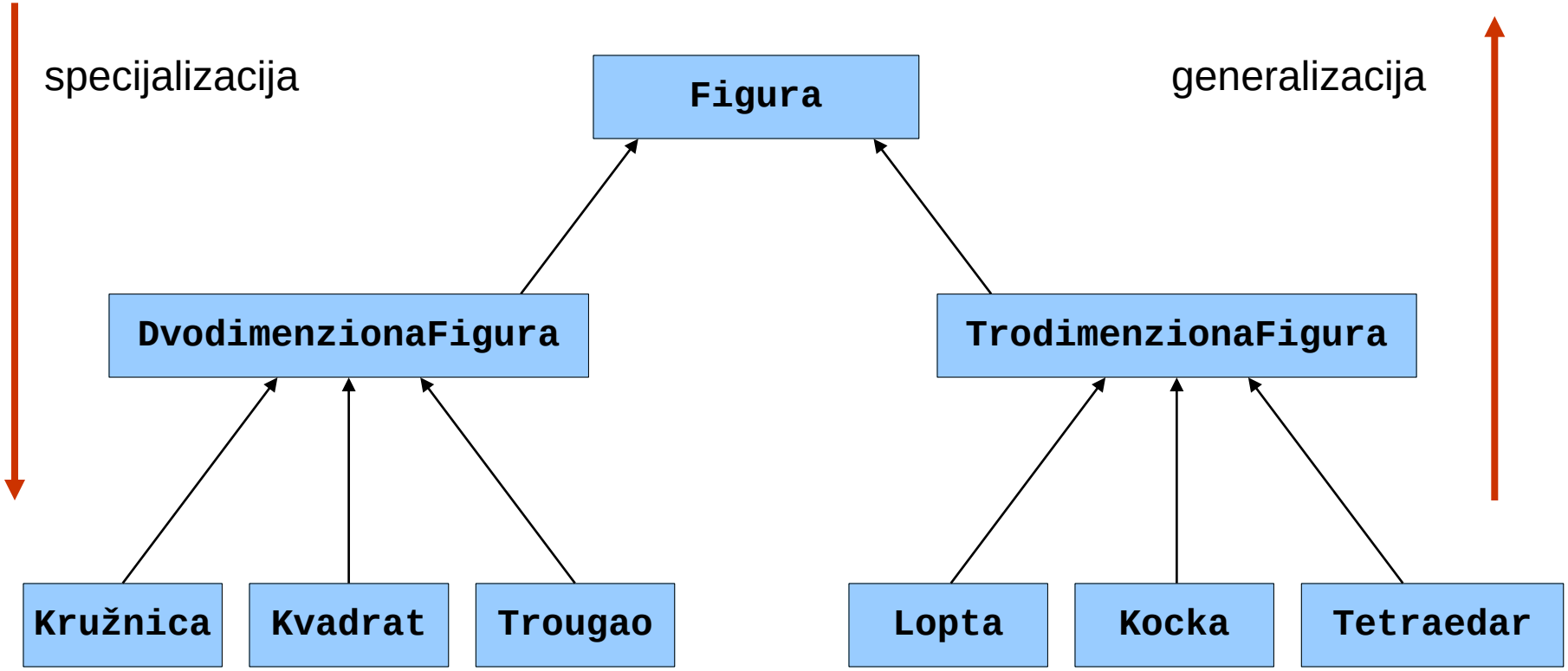
```
class complex {
  //...
public:
  complex operator!();                          // članica unarni operator!
  friend complex operator~(complex);            // globalna f-ja, unarni operator~
  complex operator-(complex);                   // članica binarni operator-
  friend complex operator+(complex,complex); // globalna f-ja, binarni operator+
};
```

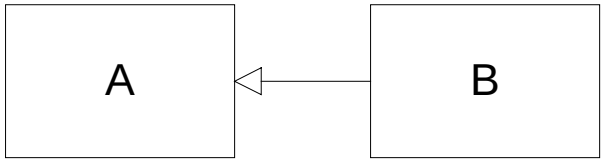
Izbor povratnih tipova operatorskih funkcija

- operatori koji menjaju levi operand (npr. =) treba da vrate referencu na levi operand.
- operatori koji biraju ili isporučuju operande (npr. [], ->, ()) obično treba da vrate referencu na izabranu vrednost.
- operatori koji izračunavaju novu vrednost, a ne menjaju svoje operande (operatori nad bitovima, +, -, *, /, %, &, \, ^, &&, ||, unarni (-, +), !~) obično treba da vrate kopiju lokalno kreiranog objekta.
- preinkrement i predekrement - obično treba da vrate referencu na operand.
- postinkrement i postdekrement - obično treba da sačuvaju kopiju svog operanda, izmene operand i zatim vrate kopiju.

Izvođenje, nasleđivanje

- Često se sreće slučaj da je jedna klasa objekata (klasa B) podvrsta (*a-kind-of*) neke druge klase (klasa A).
- **Primeri:**
  - Sisari su klasa koja je okarakterisana načinom reprodukcije.
    Mesožderi su vrsta sisara koja se hrani mesom.
    Biljojedi su vrsta sisara koja se hrani biljkama.
  - Geometrijske figure u ravni su klasa koja je okarakterisana koordinatama težišta.
    Krug je vrsta figure u ravni koja je okarakterisana dužinom poluprečnika.
    Kvadrat je vrsta figure u ravni koja je okarakterisana dužinom ivice.
- Objekti klase B imaju sve osobine klase A i još neke specijalne, sebi svojstvene.
- Specijalnija klasa B se izvodi iz generalnije klase A.
- Klasa B nasleđuje osobine klase A pa se ovakva relacija između klasa naziva *nasleđivanje* (engl. *inheritance*).
- Relacija nasleđivanja se najčešće prikazuje (usmerenim acikličnim) grafom:

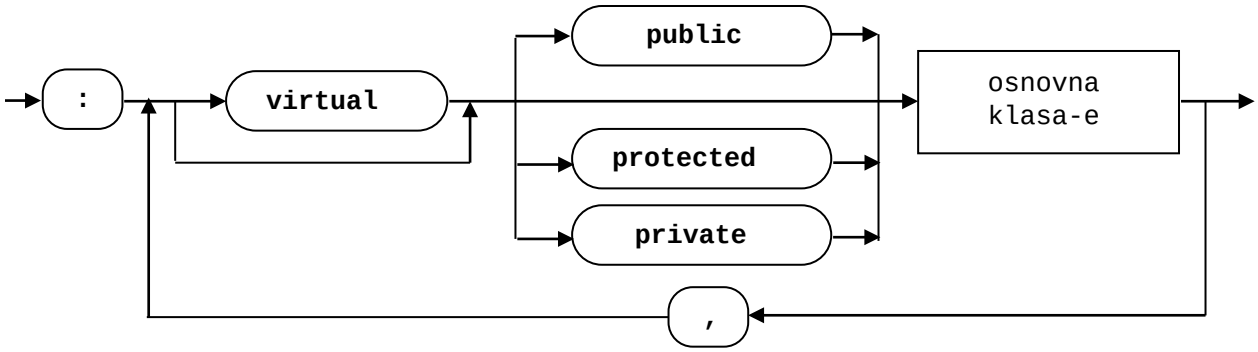

specijalizacija
generalizacija
Figura
DvodimenzionaFigura
TrodimenzionaFigura
Kružnica
Kvadrat
Trougao
Lopta
Kocka
Tetraedar



- Ako je klasa B nasledila klasu A (izvedena iz klase A), kaže se još da je:
  - klasa A *osnovna klasa* (engl. *base class*), a klasa B *izvedena klasa* (engl. *derived class*);
  - klasa A nadklasa (engl. *superclass*), a klasa B podklasa (engl. *subclass*);
  - klasa A roditelj (engl. *parent*), a klasa B dete (engl. *child*).

Definisanje izvedene klase

- Izvedena klasa se definiše navođenjem sledeće konstrukcije između identifikatora klase i znaka {:



```
class Osnovna {
    int i;          // privatni podatak clan osnovne klase

 public:
   void f();        // javna funkcija clanica osnovne klase
};
```

```
class Izvedena : public Osnovna {
    int j;          // privatni podatak clan izvedene klase

 public:
   void g();        // javna funkcija clanica izvedene klase
};
```

```
void main () {
    Osnovna b;  Izvedena d;
    b.f();
    b.g();      // ! GRESKA: g je funkcija izvedene klase, a b je objekat osnovne
    d.f();      // objekat izvedene klase d ima i funkciju f,
    d.g();      // i funkciju g
}
```

- Izvedena klasa ne nasleđuje funkciju članicu operator=, konstruktore ni destruktore osnovne klase.

- Funkcija članica operator=, podrazumevani i kopi konstruktor, kao i destruktor se generišu pri čemu:

  - Funkcija članica operator= vrši dodelu član po član;

  - Podrazumevani konstruktor ima prazno telo;

  - Kopirajući konstruktor vrši dodelu vrednosti član po član svog argumenta objektu koji se konstruiše;

  - Destruktor ima prazno telo.

- Izvedena klasa može biti osnovna klasa za sledeće izvođenje

Vidljivost i prava pristupa

- Redefinisanje identifikatora iz osnovne klase unutar izvedene klase sakriva identifikator iz osnovne klase.
- Pristup sakrivenom članu iz osnovne klase unutar izvedene klase je moguć: <osnovna klasa>::<član>.
- Izvedena klasa nema prava pristupa privatnim članovima osnovne klase.
- Labela protected: označava deo klase koji je pristupačan kako članicama tako i funkcijama izvedenih klasa.
- Članovi u ovoj sekciji se nazivaju zaštićenim članovima (engl. *protected members*).

```
class Osnovna {
  int pb;
protected:
  int zb;
public:
  int jb;
};
```

```
class Izvedena : public Osnovna {
public:
  void write(int x) {
    jb=zb=x;  // moze da pristupi javnom i zasticenom clanu,
    pb=x;     // ! GRESKA: privatnom clanu se ne moze pristupiti
  }
};
```

```
void f() {
  Osnovna b;  b.zb=5;  // odavde ne moze da se pristupa zasticenom clanu
}
```

Načini izvođenja

Pristup osnovnoj klasi	Pristupačnost u privatno izvedenoj klasi	Pristupačnost u zaštićeno izvedenoj klasi	Pristupačnost u javno izvedenoj klasi
Nepristupačan	Nepristupačan	nepristupačan	nepristupačan
Privatni	Nepristupačan	nepristupačan	nepristupačan
Zaštićeni	Privatni	zaštiđeni	zaštićeni
Javni	Privatni	zaštićeni	javni

Konstruktori i destruktori izvedenih klasa

- Prilikom kreiranja objekta izvedene klase, poziva se konstruktor te klase, ali i konstruktor osnovne klase.
- U zaglavlju definicije konstruktora izvedene klase, u listi inicijalizatora, moguće je navesti i inicijalizator osnovne klase (argumente poziva konstruktora osnovne klase).
- To se radi navođenjem imena osnovne klase i argumenata poziva konstruktora osnovne klase

```
class Osnovna {
  int bi;
public:
  Osnovna(int);  // konstruktor osnovne klase
};
Osnovna::Osnovna (int i) : bi(i) {/*...*/}
```

```
class Izvedena : public Osnovna {
  int di;
public:
  Izvedena(int);
};
```

```
Izvedena::Izvedena (int i) : Osnovna(i),di(i+1)
{/*...*/}
```

- Pri kreiranju objekta izvedene klase redosled poziva konstruktora je sledeći:
- inicijalizuje se podobjekat osnovne klase, pozivom konstruktora osnovne klase;
- inicijalizuju se podaci članovi, eventualno pozivom njihovih konstruktora, po redosledu deklarisanja;
- izvršava se telo konstruktora izvedene klase.
- Pri uništavanju objekta, redosled poziva destruktora je uvek obratan.

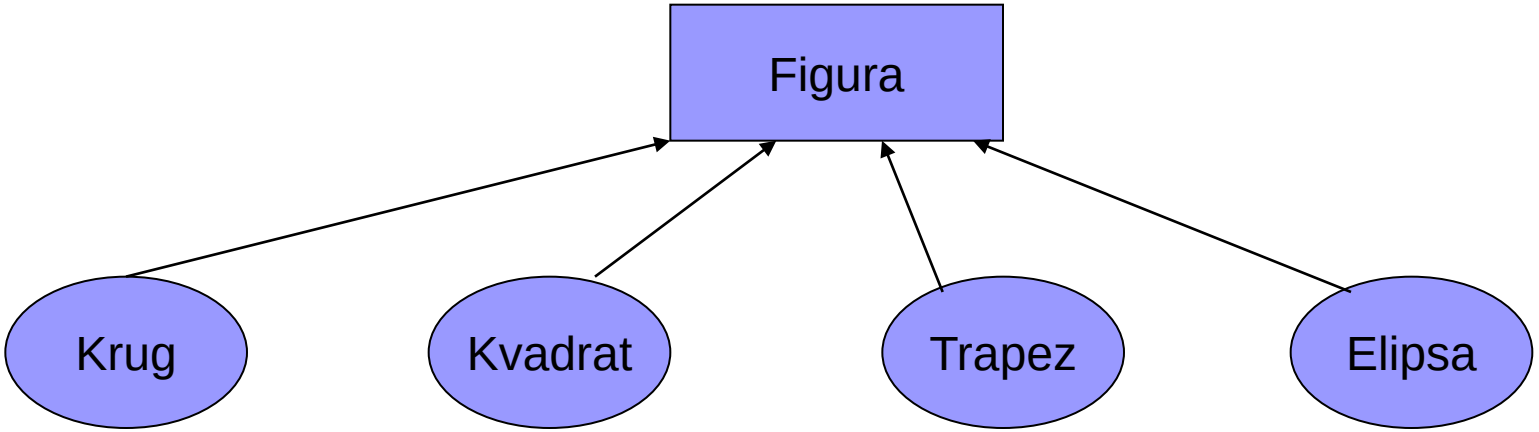
```
class Element {
public:
                        Element() {cout<<"Konstruktor klase Element.\n";}
                        ~ Element() {cout<<"Destruktor klase Element.\n";}
};

class Osnovna {
public:
                        Osnovna() {cout<<"Konstruktor osnovne klase."<<endl;}
                        ~Osnovna() {cout<<"Destruktor osnovne klase."<<endl;}
};

class Izvedena : public Osnovna {
        Element x;
public:
        Izvedena() {cout<<"Konstruktor izvedene klase."<<endl;}
        ~Izvedena() {cout<<"Destruktor izvedene klase."<<endl;}
};

void main () {
        Izvedena d;
}

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

```
#include "iostream.h"

class Student;

class Person {
 public:
       Person(){cout << "constructor Person" << endl;}
       ~Person(){cout << "destructor Person" << endl;}
};

const Student& returnPerson(const Student& p){return p;}

class Student : public Person {
 public:
       Student(){cout << "constructor Student" << endl;}
       ~Student(){cout << "destructor Student" << endl;}
};

Student returnStudent(Student s){ return s;}
```

```
class PhDStudent : public Student {
 public:
       PhDStudent(){cout << "constructor PhDStudent" << endl;}
       ~PhDStudent(){cout << "destructor PhDStudent" << endl;}
};

Student returnPhDStudent(Student s)
{
       return s;
}


int main(int argc, char* argv[])
{
       PhDStudent laza;
       returnPhDStudent(returnStudent(returnPerson(laza)));
}
```

Objekat javno izvedene klase se može posmatrati kao objekat osnovne klase.

Polimorfizam




```
void drawFigures () {
  for (int i=0; i< NoFig; i++)
    FigArray[i]->draw();
}
```

Konverzija pokazivača i referenci

- Posledice činjenice da se objekat javno izvedene klase može smatrati i objektom osnovne klase su:
  1. Pokazivač na objekat izvedene klase se može implicitno konvertovati u pokazivač na objekat osnovne klase.
  2. Isto važi i za reference.
  3. Objekat osnovne klase može se inicijalizovati objektom izvedene klase.
  4. Objektu osnovne klase može se dodeliti objekat izvedene klase bez eksplicitne konverzije.
- Pokazivač na objekat privatno izvedene klase se može implicitno konvertovati u pokazivač na objekat osnovne klase samo unutar izvedene klase.
- Pokazivač na objekat osnovne klase se može samo eksplicitno konvertovati u pokazivač na objekat izvedene klase.

Virtuelne funkcije

- Funkcije članice osnovne klase koje se u izvedenim klasama mogu redefinisati, a ponašaju se polimorfno, nazivaju se *virtuelne funkcije* (engl. *virtual functions*).
- Virtuelna funkcija se u osnovnoj klasi deklariše pomoću ključne reči `virtual` na početku deklaracije.
- Prilikom deklarisanja virtuelnih funkcija u izvedenim klasama ne mora se stavljati reč `virtual`.
- Pozivom preko pokazivača na osnovnu klasu izvršava se ona funkcija koja pripada klasi pokazanog objekta.

```
class ClanBiblioteke {
protected:
  Racun r;
public:
  virtual int platiClanarinu ()                    // virtuelna funkcija osnovne klase
    { return r-=CLANARINA; }
};

class PocasniClan : public ClanBiblioteke {
public:
  int platiClanarinu () { return r; }      // virtuelna funkcija izvedene klase
};

void main () {
  ClanBiblioteke *clanovi[100];
  //...
  for (int i=0; i<brojClanova; i++)
    cout<<clanovi[i]->platiClanarinu();
}
```

Virtuelni mehanizam se aktivira samo ako se objektu pristupa preko reference ili pokazivača.

```
class Osnovna { public: virtual void f(); };

class Izvedena : public Osnovna { public: void f(); };

void g1(Osnovna b)   { b.f(); }
void g2(Osnovna *pb) { pb->f(); }
void g3(Osnovna &rb) { rb.f(); }

void main () {
  Izvedena d;
  g1(d);                                    // poziva se Osnovna::f
  g2(&d);                                   // poziva se Izvedena::f
  g3(d);                                    // poziva se Izvedena::f
  Osnovna *pb=new Izvedena; pb->f();        // poziva se Izvedena::f
  Osnovna &rb=d; rb.f();                    // poziva se Izvedena::f
  Osnovna b=d; b.f();                       // poziva se Osnovna::f
  delete pb; pb=&b; pb->f();                // poziva se Osnovna::f
}
```

- Mehanizam koji obezbeđuje da se funkcija koja se poziva određuje po tipu objekta, a ne po tipu pokazivača ili reference na taj objekat, naziva se *dinamičko vezivanje*.
- Odlučivanje koja će se virtuelna funkcija pozvati obavlja se u toku izvršavanja programa - dinamički.
- Bitna je razlika u odnosu na mehanizam preklapanja imena funkcija koji je statički.
- Virtuelna funkcija osnovne klase ne mora da se redefiniše u svakoj izvedenoj klasi, tada važi funkcija iz osnovne klase.
- Deklaracija virtuelne funkcije u izvedenoj klasi mora da se potpuno slaže sa deklaracijom iste u osnovnoj klasi.
- Ako se u izvedenoj klasi deklariše neka funkcija koja ima isto ime kao i virtuelna funkcija iz osnovne klase, ali različit broj i/ili tipove argumenata, onda ona sakriva *sve* ostale funkcije sa istim imenom iz osnovne klase.
  - U izvedenoj klasi treba ponovo definisati sve ostale funkcije sa tim imenom.
  - Nije dobro da izvedena klasa sadrži samo neke funkcije iz osnovne klase: ne radi se o pravom nasleđivanju. Korisnik izvedene klase očekuje da će ona ispuniti sve zadatke koje može i osnovna klasa.
- Virtuelne funkcije moraju biti nestatičke članice svojih klasa, a mogu biti prijatelji drugih klasa.

Čiste virtuelne funkcije i apstraktne klase

- Virtuelna funkcija koja nije definisana za osnovnu klasu naziva se *čistom virtuelnom funkcijom*.
- Deklaracija čiste virtuelne funkcije u osnovnoj klasi sadrži umesto tela `=0`.
- Klasa koja sadrži barem jednu čistu virtuelnu funkciju naziva se *apstraktnom klasom*
- Apstraktna klasa ne može imati instance (objekte), već se iz nje samo mogu izvediti druge klase.
- Mogu da se formiraju pokazivači i reference na apstraktnu klasu.
- Pokazivači i reference na apstraktnu klasu mogu da ukazuju samo na objekte izvedenih konkretnih klasa.
- Apstraktna klasa predstavlja samo generalizaciju izvedenih klasa.
- Primer: svi izlazni, znakovno orijentisani uređaji računara imaju funkciju za ispis jednog znaka, ali se u osnovnoj klasi izlaznog uređaja ne može definisati način ispisa, jer je to specifično za svaki uređaj.
- Ako se u izvedenoj klasi ne navede definicija neke čiste virtuelne funkcije iz osnovne klase, i ova izvedena klasa je takođe apstraktna.

```
class Osnovna {
public:
  virtual void cvf () =0;                 // cista virtuelna funkcija
  virtual void vf ();                     // obicna virtuelna funkcija
};

class Izvedena : public Osnovna {
public:
  void cvf();
};

void main () {
  Izvedena izv, *pi=&izv;
  Osnovna osn;                             // ! GRESKA: Osnovna je apstraktna klasa
  Osnovna *po=&izv;
  po->cvf();                              // poziva se Izvedena::cvf()
  po->vf();                               // poziva se Osnovna::vf()
}
```

Virtuelni destruktor

- Konstruktor ne može biti virtuelna funkcija (jer se poziva pre nego što se objekat kreira)
- Destruktor može biti virtuelna funkcija.
- Ako je destruktor virtuelna funkcija, tek u vreme izvršenja se odlučuje koji destruktor se poziva.
- Destruktor osnovne klase se uvek izvršava (ili kao jedini ili posle destruktora izvedene klase).
- Kada neka klasa ima neku virtuelnu funkciju, verovatno i njen destruktor (ako ga ima) treba da bude virtuelan.
- Unutar destruktora izvedene klase ne treba pozivati destruktor osnovne klase (on se implicitno poziva).

Konstruktor ne može da bude virtuelna funkcija !!!

```
class Osnovna { public: virtual ~Osnovna(); };
class Izvedena : public Osnovna { public: ~Izvedena(); };
class OsnovnaNV { public: ~OsnovnaNV(); };
class IzvedenaNV : public OsnovnaNV { public: ~IzvedenaNV(); };
void oslobodi (Osnovna *pb) { delete pb; }
void oslobodi (OsnovnaNV *pb) { delete pb; }
void main () {
  Osnovna *pb=new Osnovna; oslobodi(pb);
  Izvedena *pd=new Izvedena; oslobodi(pd); // poziva se ~Izvedena pa ~Osnovn
  OsnovnaNV *pbn=new OsnovnaNV; oslobodi(pbn);   // poziva se ~OsnovnaNV
  IzvedenaNV *pdn=new IzvedenaNV; oslobodi(pdn); // poziva se ~OsnovnaNV
}
```

Nizovi i izvedene klase

- Neka kolekcija objekata izvedene klase nije jedna vrsta kolekcije objekata osnovne klase.
- Niz objekata izvedene klase nije jedna vrsta niza objekata osnovne klase.
- Ako se niz objekata izvedene klase prenese funkciji kao niz objekata osnovne klase, može doći do greške!!!
- Da bi se poštovala semantika i iskoristio polimorfizam, kolekcije objekata treba realizovati kao kolekcije pokazivača na objekte a ne kao kolekcije objekata.

```
class Osnovna {
public: int bi;
};

class Izvedena : public Osnovna {
public: int di;
};

void f(Osnovna *b, int i) { cout << b[i].bi; }

void main () {
  Izvedena d[5];
  d[2].bi=77;
  f(d,2);            // nece se ispisati 77
}
```

- Funkcija f smatra da je dobila niz objekata osnovne klase koji su kraći od objekata izvedene klase.
- Kada joj se prosledi niz objekata izvedene klase, funkcija nema načina da to odredi.
- Nije fizički moguće u niz objekata osnovne klase smeštati objekte izvedene klase(duži)

- **Ako se računa sa nasleđivanjem, u programu ne treba koristiti nizove objekata, već nizove pokazivača na objekte.**

```
void f(Osnovna **b, int i) { cout<<b[i]->bi; }

void main () {
  Osnovna b1;
  Izvedena d1,d2;
  Osnovna *b[5];                // b se moze konvertovati u tip Osnovna**
  b[0]=&d1; b[1]=&b1; b[2]=&d2;
  d2.bi=77;
  f(b,2);                       // ispisace se 77
}
```

- **Nije dozvoljena konverzija Izvedena** u Osnovna**.**
- **Nije dozvoljeno** :

```
void main () {
  Izvedena *d[5];       // d je tipa Izvedena**
  f(d,2);               // ! GRESKA: pokusana konverzija Izvedena** u Osnovna**
}
```

Višestruko izvodjenje (nasleđivanje)

- *višestruko nasleđivanje* (*multiple inheritance*) - klasa nasleđuje osobine više osnovnih klasa.
- **Primer:** "motocikl sa prikolicom" je vrsta "motocikla", ali je i "vozilo sa tri točka". Pri tome "motocikl" nije "vozilo sa tri točka" niti je "vozilo sa tri točka" vrsta "motocikla".
- Klasa se deklariše kao naslednik više klasa tako što se u zaglavlju deklaracije navode osnovne klase.
- Ispred svake osnovne klase treba da stoji reč `public`, da bi izvedena klasa nasleđivala prava pristupa članovima.

---

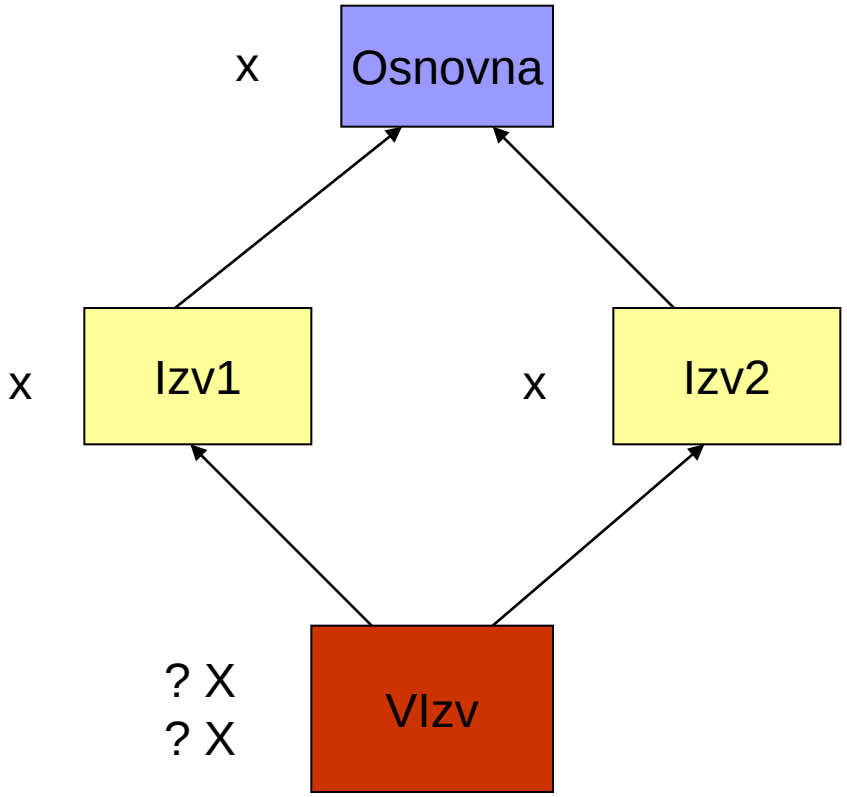
class Izvedena : **public** Osnovna1, **public** Osnovna2, **private** Osnovna3 {/* ... */};

- Može i private i protected.
- Podrazumevani: **private** za *klasu* i **public** za *struct*.

Konstruktori i destruktori – višestruko nasl.

- Sva pravila o nasleđenim članovima važe i kod višestrukog nasledjivanja.
- Konstruktori osnovnih klasa se pozivaju pre konstruktora članova izvedene klase i konstruktora izvedene klase.
- Konstruktori osnovnih klasa se pozivaju po redosledu deklarisanja.
- Destruktori osnovnih klasa se izvršavaju na kraju, posle destruktora izvedene klase i destruktora članova.

Višestruki podobjekti




```
class Osnovna {
        public: int x;
};
class Izv1 : public Osnovna {/*...*/};
class Izv2 : public Osnovna {/*...*/};
class VIzv : public Izv1, public Izv2 {
        public: int fun();
};
```

- Ako klasa ima višestruku nevirtuelnu osnovnu klasu **X** onda će objekti te klase Imati više podobjekata tipa **X**

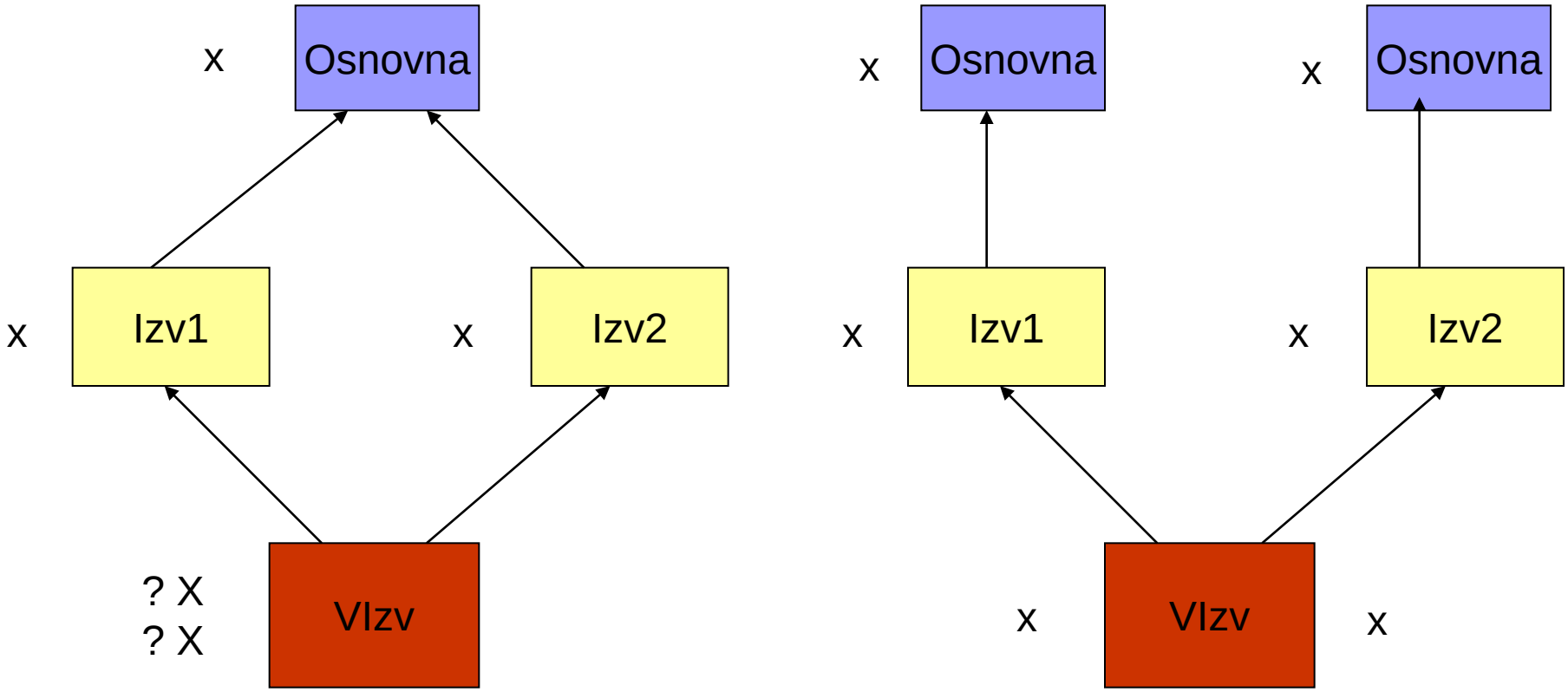
- Članovima osnovne klase **X** može se pristupiti nedvosmislenim navodjenjem njihove pripadnosti, korišćenjem operatora **: :**

Vlzv :: Izv1 :: Osnovna :: X

- Konverzija pokazivača tj reference na izvedenu klasu u pokazivač/referencu na višestruku osnovnu klasu može se izvršiti samo ako je nedvosmislena !!!

- Nedvosmislenost znači da ne postoje dva ili više entiteta koja odgovaraju navedenom imenu

```
void Vlzv :: fun()
{
    x=2;                        // dvosmislenost,  Izv1::X  ili Izv2::X  ,  Greška !!!
    Izv1::x = Izv2::x + 1;   // ispravno
}
```



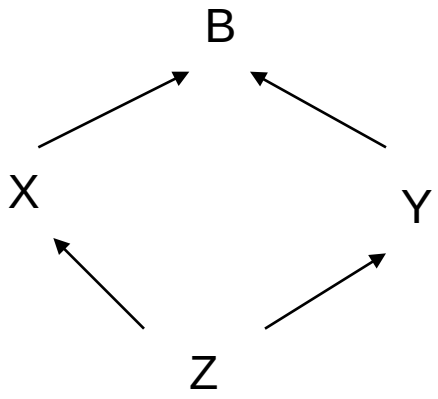
Nije dozvoljeno da jedna klasa bude višestruka direktna osnovna klasa !!!

```
class A :  public X,  public X
{ };
```

Virtuelne osnovne klase

- Osnovna klasa može biti virtuelna – treba je deklarisati kao virtuelnu sa **`virtual`**
- Ako je osnovna klasa virtuelna onda se podobjekti osnovne klase unutar objekta Izvedene klase dele sa svim podobjektima koji imaju istu osnovnu klasu kao virtuelnu
- Obraćanje članu osnovne klase nije dvosmisleno ako postoji samo jedan primerak tog člana bez obzira na način pristupa
- Ako je potrebno da izvedena klasa poseduje samo jedan podobjekat indirektne osnovne klase onda osnovnu klasu treba deklarisati kao virtuelnu !!!

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



- Sada klasa Z ima samo jedan skup članova klase B.
- Moraju se i X i Y virtuelno izvesti iz B;

  Ako je samo jedna izvedena virtuelno ostaju dva skupa članova.
- Konstruktori virtuelnih osnovnih klasa se pozivaju pre konstruktora nevirtuelnih osnovnih klasa.
- Svi konstruktori osnovnih klasa se pozivaju pre konstruktora članova i konstruktora izvedene klase.

Generički mehanizam - šabloni

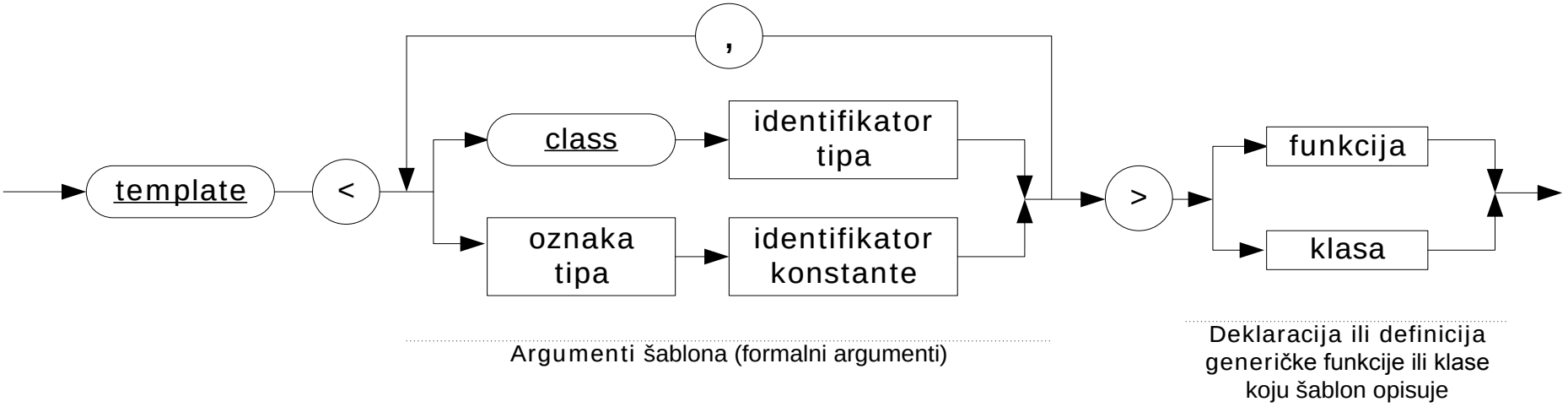
- Isti algoritam za različite podatke (sortiranje prirodnih, celih, realnih brojeva)

```
void sort(int a[], int n) { … }
void sort(float a[], int n) { … }
void sort(specKlasa a[], int n) { … }

void push(Klasa a){ … }
```

```
char   max(char i, char j){return i>j?i:j;}
int    max(int i,  int j} {return i>j?i:j;}
float  max(float i,float j}{return i>j?i:j;}
```

- U C++ postoji mogućnost definisanja šablona za opisivanje obrade za opšti slučaj, ne navodeći konkretne tipove podataka.
- Na osnovu šablona mogu da se automatski (prevodioc) generišu konkretne f-je za konkretne tipove podataka.
- Šabloni mogu da se definišu i za klase.
- Ako se koriste šabloni, onda se radi o generičkim funkcijama ili generičkim klasama na osnovu kojih se kasnije generišu konkretne funkcije ili klase.
- Slično funkcijama, ovi argumenti su formalni argumenti koji će biti zamenjeni stvarnim argumentima. Mehanizam je statički – instance šablona se prave tekstualnom zamenom u vreme prevođenja
- Sve funkcije tj klase generisane na osnovu istog šablona imaju istu realizaciju a razlikuju se po tipovima sa kojima manipulišu.
- Generičke klase predstavljaju **parametrizovane tipove**.

Definisanje šablona



- Identifikatori tipa - (formalni argumenti šablona) mogu da se koriste unutar generičke funkcije ili klase na svim mestima gde se očekuju tipovi.
- Identifikatori konstante - mogu da se shvate kao simboličke konstante i da se koriste unutar generičke funkcije ili klase svuda gde se očekuju konstante.
- Oznaka tipa - za simboličke konstante mogu da budu standardni ili korisnički tipovi podataka.
- Odvojeno prevođenje šablona nema smisla – šabloni se smeštaju u *.h datoteke i uključuju tamo gde se koriste.
- Mana šablona: pošto su u *.h datotekama – korisnik vidi celu implementaciju algoritama, a ne samo interfejs.

```
//sablon funkcije
template <class T> T max(T a, T b){return a>b?a:b;}


//sablon prototipa
template <class T> void sort(T a[ ], int n);

//sablon definicije funkcije
template <class T> void sort(T a[ ], int n){/*...*/};

//sablon klase
template <class T, int k> class Vekt{
  int duz;
  T niz[k];
public:
  Vekt();
  T& operator[](int i) const{return niz[i];}
};

template <class T, int k> Vekt<T,k>::Vekt(){
   duz=k; for(int i=0;i<k;niz[i++]=0);
}
```

Generisanje funkcija

Funkcije na osnovu zadatog šablona se generišu:

1. automatski - kad se naiđe na poziv generičke f-je sa stvarnim argumentima koji mogu da se uklope u šablon bez konverzije tipova argumenata.
2. na zahtev programera - navođenjem deklaracije (prototipa) za datu generičku f-ju sa željenim argumentima.

- Kada naiđe na poziv funkcije za koju ne postoji definicija, a postoji odgovarajuća generička funkcija, prevodilac generiše odgovarajuću definiciju funkcije.
- "Odgovarajuća generička funkcija" znači: stvarni argumenti se bez konverzije uklapaju u formalne argumente.
- Generisanje na zahtev se postiže navođenjem prototipa sa tipovima stvarnih argumenata.
- Pri pozivanju generisane funkcije vrše se uobičajene konverzije tipova.
- Generisanje funkcije iz šablona će biti sprečeno ako se prethodno pojavi definicija odgovarajuće funkcije.
- Generisanje funkcije iz šablona se vrši samo ako odgovarajuća funkcija (generisana ili definisana) ne postoji

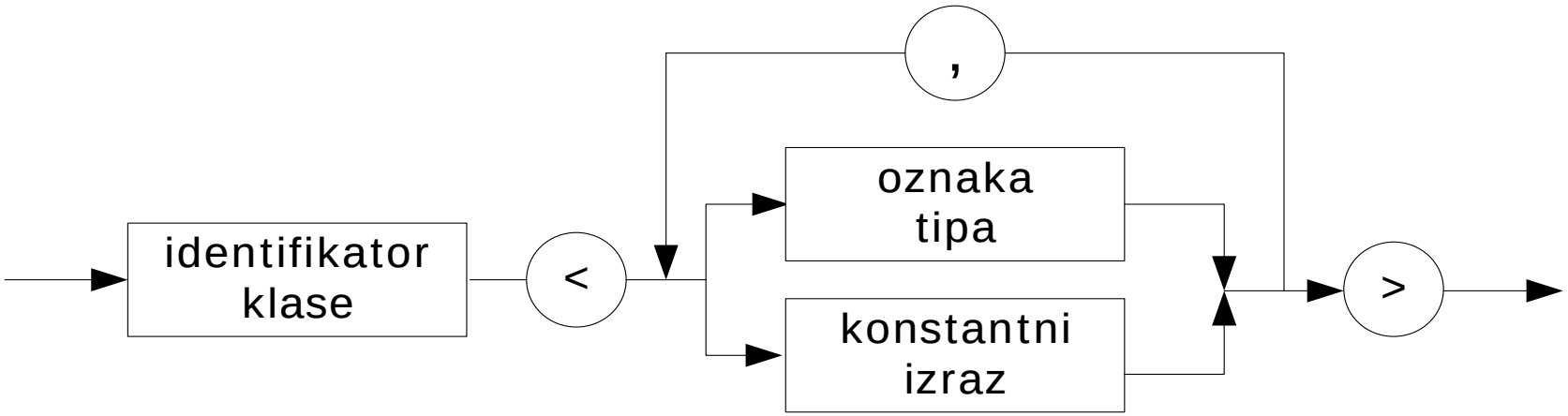
```
char *max(char *cp1,char *cp2){           // definicija obicne funkcije
  return strcmp(cp1,cp2)>=0?cp1:cp2;      // da se spreci poredjenje pokazivaca
}
void main() {
  int i1=1, i2=2; float f=0.5; char c1='a', c2='b'; char *a="alfa", *b="beta";
  int i=max(i1,i2);                       // generise se int max(int,int)
  char c=max(c1,c2);                      // generise se char max(char,char)
  int g=max(i1,c1);                       // ! GRESKA
  float max(float,float);                 // zahtev za generisanje
  float g=max(i1,f);                      // poziva se float max(float,float)
  char *v=max(a,b);                       // poziva se char* max(char*,char*)
}
```

Da bi generička funkcija mogla da se pozove za argumente i tipove, mora da se traži formiranje konkretne funkcije. Tako generisana funkcija koristi se kao i obična funkcija, pa će stvarni argumenti neodgovarajućeg tipa da se konvertuju u potreban tip pre poziva funkcije, ali samo ako postoji odgovarajuća automatska konverzija tipa.

Generisanje klasa

Formalni argumenti šablona klase mogu biti: tipovi (class T) ili prosti objekti (biće konkretizovani konstantnim izrazima)

- Konkretna klasa se generiše kada se prvi put naiđe na definiciju objekta u kojoj se koristi identifikator te klase.
- Pri generisanju klase se generišu i sve funkcije članice.
- Oznaka tipa generisane klase treba da sadrži:
  1. identifikator generičke klase i
  2. listu stvarnih argumenata za generičke tipove i konstante unutar <> iza identifikatora.
- Stvarni argument može biti:
  - oznaka tipa – zamenjuje formalni argument koji je identifikator tipa;
    oznaka tipa može biti standardni tip, identifikator klasa ili izvedeni tip (npr. pokazivač).
  - konstantni izraz: zamenjuje formalni argument koji je identifikator konstante;
    ako izraz sadrži operator > ovaj se mora pisati u zagradama (>).



```
template <class T, int n>
        class Vektor {
                T  v[n];
public:
        T& operator[] (int i) {
                if ((i>0) && (i<n)) return v[i];
                else error("van opsega");
        }
};
```

// primerci ove klase sadrže niz celih brojeva od 10 komponenti

```
        Vektor <int, 10> niz1;
```

// primerci ove klase sadrže niz realnih brojeva u dvostrukoj tačnosti od 20 komponenti

```
        Vektor <double, 20> niz2;
```

- Deklaracija šablona može biti samo globalna
- Svaka upotreba imena šablonske klase predstavlja deklaraciju konkretne šablonske klase
- Poziv šablonske funkcije ili uzimanje njene adrese predstavlja deklaraciju konkretne šablonske funkcije
- Ako se definiše obična negenerička klasa ili funkcija sa istim imenom i sa potpuno odgovarajućim tipovima kao što je šablon klase tj funkcije, onda ova definicija predstavlja definiciju konkretne šablonske klase tj funkcije za date tipove

```
template<class T>                    // sablon
       void sort(vector<T>& v) { … }


void sort(vector<int>& v) { … }  // konkretna funkcija
```

- Funkcija članica šablonske klase je implicitno šablonska funkcija
- Prijateljske funkcije šablonske klase nisu implicitno šablonske funkcije

ZADATAK: Sastaviti genericku klasu na C++ -u za statičke stekove zadatih kapaciteta. U glavnom programu prikazati mogućnosti te klase (rad steka ne zavisi od tipa objekta koji se smeštaju na stek).

```
// stek.h
template <class Pod, int kap>
class Stek
{
        Pod stek[kap]; //Stek u obliku niza
        int vrh;  //Indeks vrha steka
        int gre;  //Indikator greske
        public:
                Stek(){vrh=gre=0;}  //Inicijalizacija
                void stavi(const Pod &pod);
                void uzmi (Pod &pod);
                int vel() const {return vrh;}  //Broj podataka na steku
                void brisi() {vrh=gre=0;}
                int prazan() const {return vrh==0;}
                int pun() const {return vrh==kap;}
                int greska() const {return gre;}
};
```

```
template <class Pod, int kap>
void Stek<Pod,kap>::stavi(const Pod &pod)
{
        if (!(gre=vrh==kap))
                stek[vrh++]=pod;
}


template <class Pod, int kap>
void Stek<Pod,kap>::uzmi(const Pod &pod)
{
        if (!(gre=vrh==0))
                pod=stek[--vrh];
}
```

```
// stek.cpp
#include <stek.h>
#include <iostream.h>
const int CVEL=10; DVEL=3;
void main() {
        int i;
        double d;
        Stek <int, CVEL> clb_stek;          //Stek sa celobrojnim podacima
        Stek <double, DVEL> dbl_stek;       //Stek sa realnim podacima
        while(!clb_stek.pun()) {
                cin>>i;
                clb_stek.stavi(i);
                cout<<i; }
        while(!clb_stek.prazan()) {
                clb_stek.uzmi(i);
                cout<<i; }
        while(!dbl_stek.pun()) {
                cin>>d;
                dbl_stek.stavi(d); }
        while(!dbl_stek.prazan()) {
                dbl_stek.uzmi(d);
                cout<<d; }
}
```

Zadatak: Sastaviti generičku funkciju na C++ jeziku za nalaženje ‘fuzije’ dva uređena niza objekata u treći, na isti način uređen niz.
Sastaviti glavni program koji korišćenjem predhodne funkcije nalazi fuziju nizova:

a) celih brojeva uređenih po vrednosti.
b) tačaka u ravni uređenih po odstojanjima od koordinatnog početka.
c) pravougaonika uređenih po površinama.

```
#include <iostream.h>
template <class T>
void fuzija(T a[ ], int na, T b[ ], int nb, T c[ ])
{
      for(int ia=0, ib=0, ic=0; ia<na || ib<nb; ic++)
      c[ic]= ia==na ? b[ib++]:
                ib==nb ? a[ia++]:
                    a[ia]<b[ib] ? a[ia++]: b[ib++];
}
```

```
class Tacka{
        double x,y;
public:
  // citanje tacke
        friend istream& operator>>(istream& dd, Tacka& tt);
  // upis tacke
        friend ostream& operator<<(ostream& dd, const Tacka& tt);

        friend int operator<(const Tacka& t1, const Tacka &t2);
};

inline istream& operator>>(istream& dd, Tacka& tt)
{ return dd>>tt.x>>tt.y;}
inline ostream& operator<<(ostream& dd, Tacka& tt)
{ return dd<<"T("<<tt.x<<","<<tt.y<<")";}
inline operator<(const Tacka& t1, const Tacka& t2)
{ return t1.x*t1.x+t1.y*t1.y<t2.x*t2.x+t2.y*t2.y;}
```

```
class Pravougaonik{
       double a,b ;
public:
// citanje pravougaonika
  friend istream& operator>>(istream& dd, Pravougaonik& pp);
// upis pravougaonika
  friend ostream& operator<<(ostream& dd, const Pravougaonik& pp);
  friend int operator<(const Pravougaonik& p1,
                       const Pravougaonik& p2);
};
inline istream& operator>>(istream& dd, Pravougaonik& pp)
{ return dd>>pp.a>>pp.b;}

inline ostream& operator<<(ostream& dd, Pravougaonik& pp)
{ return dd<<"P["<<pp.a<<","<<pp.b<<"]";}

inline bool operator<(const Pravougaonik& p1, const Pravougaonik&
{ return p1.a*p1.b<p2.a*p2.b;}
```

```
void main()
{
        int na, nb, nc, i;
        cout<<”Fuzija celih brojeva:\n”;
        cin>>na;
        int *a=new int[na];
        for(i=0;i<na;i++)
                cin>>a[i];
        cin>>nb;
        int *b=new int[nb];
        for(i=0;i<nb;i++)
                cin>>b[i];
        nc=na+nb;
        int *c=new int[nc];
        fuzija(a, na, b, nb, c);
        for(i=0;i<nc;i++)
                cout<<” “<<c[i];
        delete []a;
        delete []b;
        delete []c;
```

```
cout<<"\nFuzija nizova tacaka:\n";
cin>>na;
Tacka *a=new Tacka[na];
for(i=0;i<na;i++)
      cin>>a[i];
cin>>nb;
Tacka *b=new Tacka[nb];
for(i=0;i<nb;i++)
      cin>>b[i];
nc=na+nb;
Tacka *c=new Tacka[nc];
fuzija(a, na, b, nb, c);
for(i=0;i<nc;i++)
      cout<<" "<<c[i];
delete []a;
delete []b;
delete []c;
```

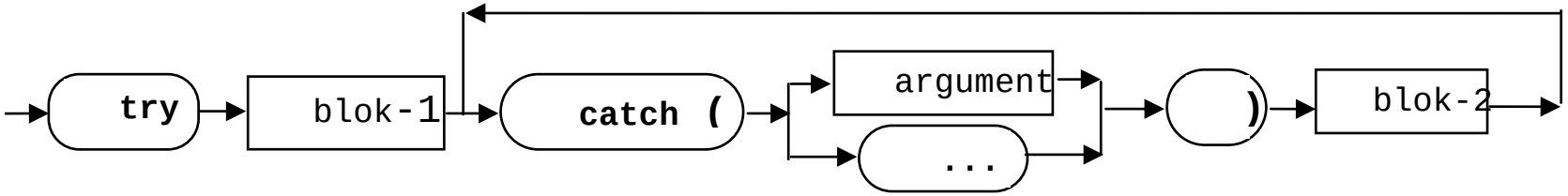
```
        cout<<"\nFuzija pravougaonika:\n";
        cin>>na;
        Pravougaonik *a=new Pravougaonik[na];
        for(i=0;i<na;i++)
              cin>>a[i];
        cin>>nb;
        Pravougaonik *b=new Pravougaonik[nb];
        for(i=0;i<nb;i++)
              cin>>b[i];
        nc=na+nb;
        Pravougaonik *c=new Pravougaonik[nc];
        fuzija(a, na, b, nb, c);
        for(i=0;i<nc;i++)
              cout<<" "<<c[i];

        delete []a;
        delete []b;
        delete []c;
}
```

Obrada Izuzetaka

- Izuzetak (*exception*) je događaj koji treba posebno obraditi izvan osnovnog toka programa.
- Na primer, greška koja se može javiti u nekoj obradi predstavlja izuzetnu situaciju.
- Ako jezik ne podržava obradu izuzetaka, dolazi do sledećih problema:
  - posle izvršenja dela programa (ili poziva funkcije) u kojem može doći do greške vrši se testiranje statusa, te se obrada greške smešta u jednu granu a obrada uobičajene situacije u drugu granu if naredbe;
  - pri lancu hijerarhijskih poziva funkcija, ukoliko grešku treba propagirati prema višem nivou, svaki nivo pozivanja treba da izvrši testiranje da li je došlo do greške i pomoću return vrati kod greške.
- C++ nudi mehanizam za efikasnu obradu izuzetaka izvan osnovnog toka kontrole:
  - izuzetak treba da bude izazvan (throw);
  - obrada se na tom mestu prekida i nastavlja u rutini (hendleru) za obradu izuzetka (catch);
  - izuzetak se smatra objektom klase (tipa) koji se dostavlja hendleru kao argument;
  - za svaki tip izuzetka se definiše poseban hendler.

Obrada izuzetaka

- Sintaksa je sledeća:



- `blok-1` je programski blok unutar kojeg mogu da se jave izuzeci.
- `argument` se sastoji od oznake tipa i identifikatora argumenta.
- `...` označavaju univerzalni hendler – on se aktivira ako ne postoji hendler sa adekvatnim tipom izuzetka.
- `blok-2` je telo hendlera.
- Definicija hendlera liči na definiciju funkcije sa tačno jednim argumentom.
- Kaže se da je hendler tipa T ako je njegov `argument` tipa T, odnosno ako obrađuje izuzetke tipa T.
- `blok-2` obrađuje izuzetke koji su se javili neposredno u `blok-1` ili u nekoj funkciji koja je pozvana iz `blok-1`.
- Nakon izvršenja `blok-2` kontrola se ne vraća na mesto gde se pojavio izuzetak.
- Unutar `blok-1` ili u pozvanim funkcijama iz `blok-1` mogu da se pojave ugnježdene naredbe `try`.
- Ako se u `blok-1` ne javi izuzetak, preskaču se svi hendleri (kontrola se prenosi na kraj naredbe `try`).

```
try {
   ...
   radi();
   ...
}

catch(const char *pz){// Obrada izuzetka tipa znakovnog niza}

catch(const int i){// Obrada izuzetka celobrojnog tipa}

catch(...){//Obrada izuzetka proizvoljnog tipa koji nije
jedan od gornjih}
```

```
1
2   // A simple exception-handling example that checks for
3   // divide-by-zero exceptions.
4   #include <iostream>
5
6   using std::cout;
7   using std::cin;
8   using std::endl;
9
10  #include <exception>
11
12  using std::exception;
13
14  // DivideByZeroException objects should be thrown by functions
15  // upon detecting division-by-zero exceptions
16  class DivideByZeroException : public exception {
17
18  public:
19
20     // constructor specifies default error message
21     DivideByZeroException::DivideByZeroException()
22        : exception( "attempted to divide by zero" ) {}
23
24  };  // end class DivideByZeroException
```

```
26  // perform division and throw DivideByZeroException object if
27  // divide-by-zero exception occurs
28  double quotient( int numerator, int denominator )
29  {
30     // throw DivideByZeroException if trying to divide by zero
31     if ( denominator == 0 )
32        throw DivideByZeroException(); // terminate function
33
34     // return division result
35     return static_cast< double >( numerator ) / denominator;
36
37  }  // end function quotient
38
39  int main()
40  {
41     int number1;    // user-specified numerator
42     int number2;    // user-specified denominator
43     double result;  // result of division
44
45     cout << "Enter two integers (end-of-file to end): ";
46
```

```
47      // enable user to enter two integers to divide
48      while ( cin >> number1 >> number2 ) {
49
50         // try block contains code that might throw exception
51         // and code that should not execute if an exception occurs
52         try {
53            result = quotient( number1, number2 );
54            cout << "The quotient is: " << result << endl;
55
56         } // end try
57
58         // exception handler handles a divide-by-zero exception
59         catch ( DivideByZeroException &divideByZeroException ) {
60            cout << "Exception occurred: "
61                 << divideByZeroException.what() << endl;
62
63         } // end catch
64
65         cout << "\nEnter two integers (end-of-file to end): ";
66
67      }  // end while
69      cout << endl;
71      return 0;  // terminate normally
72
73   }  // end main
```

Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z

Izazivanje izuzetaka

- Izazivanje (prijavljivanje, bacanje) izuzetaka se vrši naredbom:

```
throw izraz
```

gde izraz svojim tipom određuje koji hendler će biti aktiviran; vrednost izraza se prenosi hendleru kao argument.

- Izuzetak se može izazvati iz bloka ili iz bilo koje funkcije direktno ili indirektno pozvane iz bloka naredbe **`try`**.
- Funkcije iz kojih se izaziva izuzetak mogu biti i članice klasa, operatorske funkcije, konstruktori, a i destruktori.
- U deklaraciji ili definiciji funkcija može da se navede spisak tipova izuzetaka koje funkcija izaziva.
- Navođenje spiska tipova izuzetaka se postiže pomoću **`throw(niz_identifikatora)`** iza liste argumenata.
- Ako se stavi ovakva konstrukcija, a funkcija izazove izuzetak tipa koji nije u nizu identifikatora – to je greška tj nepredvidivo ponašanje.
- Ako se ništa ne navede, funkcija sme da prijavi izuzetak proizvoljnog tipa.

- Za (dinamički) ugnježdene naredbe `try` hendler unutrašnje naredbe može da izazove izuzetak.
- Takav izuzetak se prosleđuje hendleru spoljašnje naredbe `try`.
- Unutar hendlera ugnježdene naredbe `try` izuzetak može da se izazove i pomoću naredbe `throw` bez izraza.
- Takav izuzetak ima tip hendlera u kojem je izazvan.

```
void radi(...)throw(char *, int){
  if(...) throw "Izuzetak!";
  if(...) throw 100;
  if(...) throw Tacka(0,0);  // GRESKA: nije naveden tip izuzetka Tacka
}
```

Prihvatanje izuzetaka

- Hendler tipa B može da prihvati izuzetak tipa D ako:
  - B i D su istih tipova
  - B je javna osnovna klasa za klasu D
  - B i D su pokazivački tipovi i D može da se standardnom konverzijom promeni u B.
- Na mestu izazivanja izuzetka formira se privremeni objekat sa vrednošću izraza.
- Privremeni objekat se prosleđuje najbližem (prvom na koji se naiđe) hendleru.
- Ako su `try` naredbe ugnježdene, izuzetak se obrađuje u prvom odgovarajućem hendleru tekuće naredbe `try`.
- Ako se ne pronađe odgovarajući hendler - izuzetak se prosleđuje hendleru sledećeg (višeg) nivoa naredbe `try`.
- Prilikom navođenja hendlera treba se držati sledećih pravila:
  - hendlere tipa izvedenog iz neke osnovne klase treba stavljati ispred hendlera tipa te osnovne klase
  - univerzalni hendler treba stavljati na poslednje mesto.
- Predaja kontrole hendleru podrazumeva definitivno napuštanje bloka u kojem se dogodio izuzetak.
- Napuštanje bloka podrazumeva uništavanje svih lokalnih objekata u tom bloku i ugnježdenim blokovima.

- Objekat koji se pojavi kao operand naredbe `throw` se uništava prvi, ali se zato prethodno kopira u privremeni.
- Privremeni objekat će se uništiti tek po napuštanju tela hendlera.
- Ako se izuzetak iz hendlera H1 prosleđuje hendleru višeg nivoa H2, privremeni objekat živi do kraja hendlera H2.
- Nije dobro da tip izuzetka bude pokazivački tip, jer će se pokazivani objekat uništiti pre dohvatanja iz hendlera.

Neprihvaćeni izuzeci

- Ako se za neki izuzetak ne pronađe hendler koji može da ga prihvati
- kada se detektuje poremecen stek poziva
- kada se u destruktoru, u toku odmotavanja steka, postavi izuzetak

izvršava se sistemska funkcija:

```
void terminate();
```

- Podrazumeva se da ova funkcija poziva funkciju `abort()` koja kontrolu vraća operativnom sistemu.
- Ovo se može promeniti pomoću funcije `set_terminate`.
- Njoj se dostavlja pokazivač na funkciju koju treba da pozove funkcija `terminate` umesto funkcije `abort`.
- Pokazivana funkcija mora biti bez argumenata i bez rezultata (`void`).
- Vrednost funkcije `set_terminate` je pokazivač na staru funkciju koja je bila pozivana iz `terminate`.
- Iz korisničke funkcije `(*pf)` treba pozvati `exit()` za povratak u operativni sistem.
- Pokušaj povratka sa `return` iz korisničke funkcije `(*pf)` dovešće do nasilnog prekida programa sa `abort()`.

```
typedef void (*PVF) ();
PF set_terminate(PVF pf);
```

- Ako se u nekoj funkciji izazove izuzetak koji nije na spisku naznačenih izuzetaka, izvršava se funkcija:

```
void unexpected();
```

- Podrazumeva se da ova funkcija poziva funkciju `terminate()`.
- Ovo se može promeniti pomoću funcije **`set_unexpected.`**
- Njoj se dostavlja pokazivač na funkciju koju treba da pozove funkcija `unexpected` umesto `terminate`.
- Pokazivana funkcija mora biti bez argumenata i bez rezultata (`void`).
- Vrednost funkcije `set_unexpected` je pokazivač na staru funkciju koja je bila pozivana iz `unexpected`.
- Pokušaj povratka sa `return` iz korisničke funkcije `(*pf)` dovešće do nasilnog prekida programa sa `abort()`.

```
typedef void (*PF) ();
PF set_unexpected(PF pf);
```

```
void fun() throw (iz1,  iz2) {
…
}

ili

void f() {
       try {
       …
       }
       catch (iz1) {throw;}
       catch (iz2) {throw;}
       catch (…) {unexpected();}
}
```

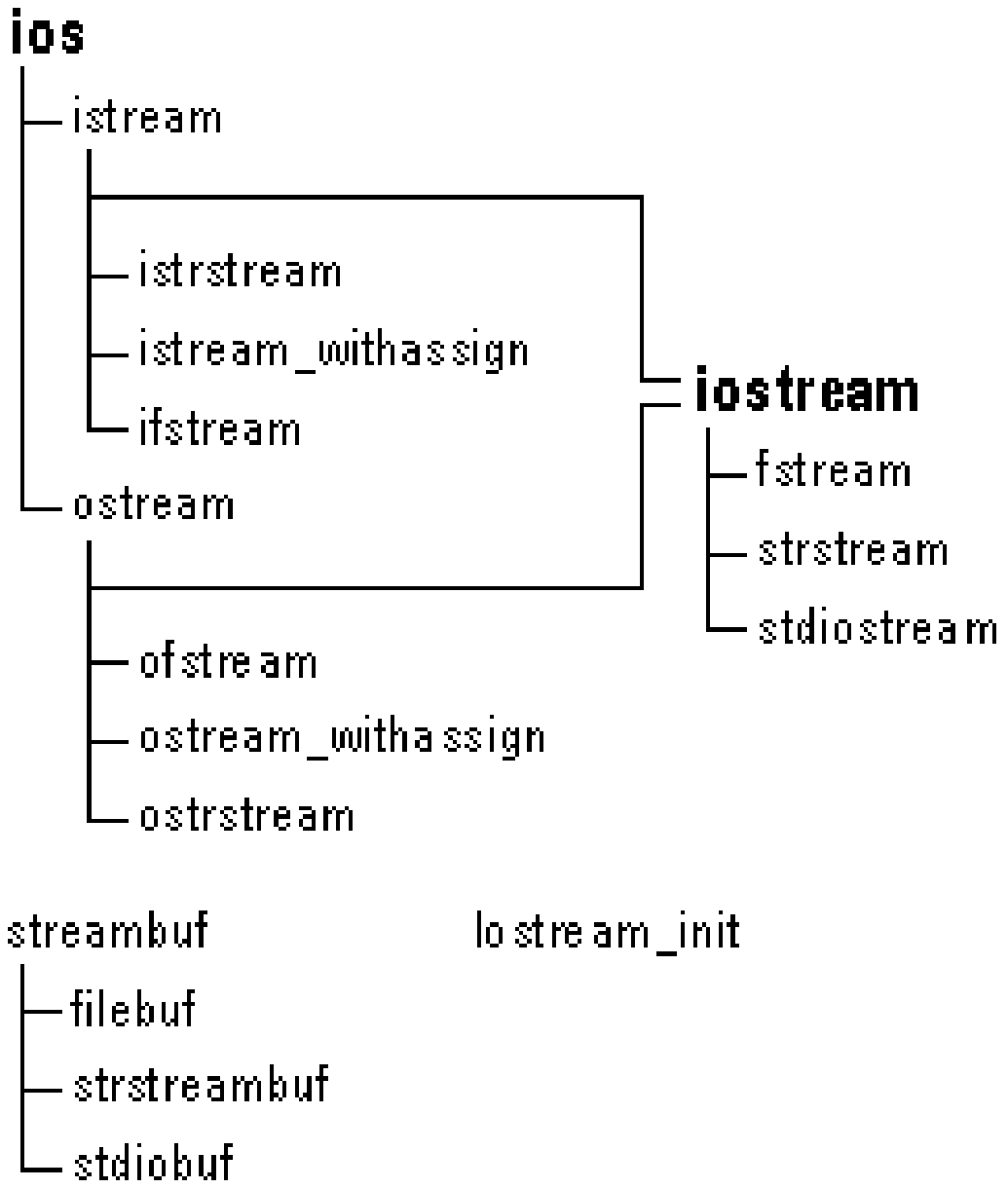
Ulazno / izlazni tokovi

Ulazno/Izlazni – Tokovi

- Nasledje iz C-a: Ne postoji ugradjena naredba tj operacija za ulaz i izlaz.
- Sve operacije za ulaz/izlaz su realizovane kao bibliotečke funkcije.
- C – standardne funkcije za ulaz-izlaz (u <stdio.h>):
  - `int printf( char* format, …)`
  - `int scanf(char* format, …)`
- C++ :
  - <stdio.h>
    - `int printf(const char* format, …)`
    - `int scanf(const char* format, …)`
  - biblioteka klasa za objektno orijentisanu realizaciju ulaza i izlaza (efikasniji način)

- Tok – logički koncept koji predstavlja sekvencijalni ulaz ili izlaz znakova na neki uredjaj ili datoteku.
- Datoteke u C++-u su samo dugački nizovi bajtova i nazivaju se tokovima. Ne postoji suštinska razlika između toka na disku (datoteke) i toka u operativnoj memoriji.
- Rad sa tokovima u C++ se realizuje odgovarajućim klasama. Konkretni tokovi su primerci (objekti) tih klasa.
- Većina operacija nad tokovima je ista, bez obzira gde su oni smešteni.

Hijerarhija klasa za realizaciju ulazno-izlaznih operacija



- Dve osnovne klase **(<iostream.h>)**
  - **istream** - ulaz
  - **ostream - izlaz**
- Dve najvažnije klase koje se izvode iz klase **iostream** su:
  - **fstream**, za rad sa datotekama
    - ifstream - ulaz
    - ofstream - izlaz
    - fstream – ulaz i izlaz
  - **strstream**, za rad sa nizovima (eng. *strings*, tj. tokovima u operativnoj memoriji)
    - istrstream - za uzimanje podataka iz tokova
    - ostrstream - za smeštanje podataka u tokove
    - strstream - za uzimanje i smeštanja podataka u/iz tokova

Standardni tokovi

- Postoje 4 standardna toka (globalni statički objekti) :
  - **cin** – glavni (standardni) ulaz tipa istream (objekat klase istream_withassign) . Standardno predstavlja tastaturu, ukoliko se drugačije ne specificira (da se izvrši skretanje glavnog ulaza unutar samog programa ili u komandi operativnog sistema za izvršavanje programa).
  - **cout** – glavni (standardni) izlaz tipa ostream (objekat klase ostream_withassign). Predstavlja ekran, koristi se za ispisivanje podataka koji čine rezultate izvršavanja programa.
  - **cerr** – standardni izlaz za poruke tipa ostream. Predstavlja ekran, obično se koristi za ispisivanje poruka o greškama.
  - **clog** – standardni izlaz za zabeleške tipa ostream. Predstavlja ekran, koristi se za “vođenje evidencije” o događajima za vreme izvršenja programa.

- Klasa istream ima po jednu funkciju članicu **operator>>** za sve ugradjene tipove koja služi za ulaz podataka

```
istream& istream::operator>> (tip &T)
```

tip - je ugradjeni tip objekta koji se čita

-Klasa ostream ima po jednu funkciju članicu **operator<<** za sve ugradjene tipove koja služi za izlaz podataka

```
ostream& ostream::operator<< (tip T)
```

tip - je ugradjeni tip objekta koji se ispisuje

- Biblioteka “stdio.h” jezika C i dalje može da se koristi za ulaz/izlaz podataka.
- Klase C++ su efikasnije.
- Nikako se ne preporučuje da se koriste obe vrste ulazno-izlaznih biblioteka

Klase za ulazne tokove

- Tri najvažnije klase za ulazne tokove su :
  - **istream**,
  - **ifstream**,
  - **istrsteam**.
- Klasa **istream** je najbolja za rad sa sekvencijalnim tekstualnim ulazom.
- Klasa **ifstream** podržava ulaz iz datoteke.

Konstruisanje objekata ulaznih tokova

- **Konstruisanje objekata ulaznih tokova**
  - Ako se koristi **cin** objekat, ne treba da se napravi ulazni tok.
  - Ulazni tok se mora napraviti ukoliko se koristi:
    - tok iz datoteke (file stream)
    - tok iz niza (string stream)

Konstruktori ulaznih tokova datoteka

- Postoji tri načina da se napravi ulazni tok vezan za neku datoteku:
  - korišćenjem konstruktora bez argumenata
  - konstruktor sa argumentima tj navođenjem naziva datoteke i odgovarajućih parametara
  - navođenjem deskriptora datoteke

Konstruktor bez argumenata

- Korišćenjem konstruktora bez argumenata kreira se objekat klase **ifstream**, a zatim se poziva funkcija **open** koja otvara navedenu datoteku:

```
ifstream f;
f.open ( “imedatoteke.txt", iosmod);
```

- ili

```
ifstream* f = new ifstream;
f->open(“imedatoteke.txt", iosmode);
```

Konstruktor sa argumentima

- Navođenje naziva datoteke i odgovarajućih parametara (mode flags) se vrši na sledeći način:

```
ifstream f(“imedatoteke.txt", iosmode);
```

Navođenje deskriptora datoteke

- Navođenje deskriptora datoteke vrši se za datoteku koja je već otvorena. Standardni (default) način za postizanje navedenog je:

```
int fd = _open(“datot.txt", dosmode);
ifstream f(fd);
```

Za parametre **iosmode** i **dosmode** koriste se sledeće vrednosti:

ios::app	upisivanje na kraj datoteke
ios::ate	pozicioniranje na kraj datoteke posle otvaranja
ios::in	otvara ulaznu datoteku
ios::out	otvara izlaznu datoteku
ios::nocreate	otvara datoteku ukoliko već postoji
ios::noreplace	otvara datoteku ukoliko već ne postoji
ios::trunc	otvara datoteku i briše stari sadržaj
ios::binary	binarna datoteka. Ako se ništa ne kaže, podrazumeva se rad sa tekstualnom datotekom.

Konstruktori ulaznih tokova

- Konstruktor ulaznog toka niza zahteva adresu prethodno alocirane i inicijalizovane memorije:

```
char s[] = "23.12";
double broj;
istrstream myString( s );
myString >> broj; // broj = 23.12
```

Operacije ulaznog toka

- Operator ekstrakcije (**>>**) je programiran za sve standardne C++ tipove podataka, i predstavlja najlakši način da se preuzmu bajtovi iz objekta ulaznog toka.

```
char ime[20];
cin >> ime;
```

- Operator >> pamti ulazne podatke samo do prvog unetog blanko znaka.

Operacija **get**

- Operacija **get** se ponaša kao i operator >>, osim što pamti i blanko znakove. Postoji više prototipova ove funkcije. Najčešće korišćeni su:
- **`istream &get(char& znak);`** - uzima sledeći znak iz ulaznog toka i smešta ga u promenljivu znak (npr. **`cin.get(c)`**).
- **`istream &get(char* niz, int max)`**; - čita max broj znakova (ukoliko postoji toliko znakova) i smešta ih u niz (npr. **`cin.get(ime, 20)`**).
- **`istream &get(char* niz, int max, char kraj)`**; - čita sve znakove do prvog pojavljivanja znaka kraj. Pored toga, može da pročita najviše max broj znakova (npr. **`cin.get(ime, 20, '\n')`**).
- Operacija **getline** je veoma slična operaciji **get**, jedino što uklanja znak prekida (npr. '\n'), dok **get** ne uklanja.

Operacija **read**

- Operacija **read** čita bajtove iz toka u određeni deo memorije. Mora se navesti lokacija gde se upisuju pročitani podaci, kao i broj pročitanih bajtova. Za citanje sloga

```
struct Radnik
{
   char ime[20];
   double plata;
};
```

- iz datoteke “datoteka.dat":

```
ifstream is( “datoteka.dat", ios::binary | ios::nocreate );
```

- Ukoliko je datoteka uspešno otvorena, vrši se čitanje podataka iz toka koji se upisuju u strukturu r, i odmah zatim i štampanje strukture r:

```
if( is )
{
   Radnik r;
   is.read( (char *) &r, sizeof(r) );
   cout << r.ime << ' ' << r.plata << endl;
}
```

- Na kraju otvoreni tok treba zatvoriti:

```
is.close();
```

- Ukoliko se ne navede broj bajtova koji treba pročitati, čitanje prestaje kada se dođe do kraja datoteke.

- **Operacije:**
  - **get i getline se koriste za ulazne tokove tekstualnih datoteka,**
  - **read i za ulazne tokove binarnih datoteka.**

- Tokovi datoteka pamte pokazivač na poziciju u datoteci koja će biti pročitana sledeća. Vrednost tog pokazivača se može odrediti pomoću operacije **seekg**:

```
ifstream is( "plata.dat", ios::binary | ios::nocreate );
is.seekg (8);     // 8 bajtova
```

- Vrednost pokazivača se može dobiti pomoću operacije **tellg** (npr. `is.tellg()`).

Preklapanje operatora ekstrakcije

- Preklapanjem (overloading) operatora >> za neku klasu, promeniće se funkcionalnost tog operatora u slučaju upotrebe te klase.

```
class Datum
{
public:
   int dan, mesec, godina;
   friend istream& operator>> ( istream& is, Datum& dt );

};

istream& operator>> ( istream& is, Datum& dt )
{
   is >> dt.dan >> dt.mesec >> dt.godina;
   return is;
}

Datum dt;
cin >> dt;
```

Izlazni tokovi

- Tri najvažnije klase za izlazne tokove su **ostream**, **ofstream** i **ostrstream**.
  - Veoma retko se konstruišu objekti klase **ostream**, već se koriste predefinisani objekti (**cout**, **cerr**, **clog**).
  - Klasa **ofstream** podržava rad sa datotekama.
  - Za kreiranje niza u memoriji treba koristiti klasu **ostrstream**.

- **Konstruisanje objekata izlaznih tokova**
  - Konstruisanje objekata izlaznih tokova se vrši potpuno isto kao i kod ulaznih tokova (isto postoje tri načina), jedino što se koristi klasa **ofstream**.

Upotreba operatora umetanja

- Operator umetanja (<<) je programiran za sve standardne C++ tipove podataka, i služi za slanje bajtova objektu izlaznog toka.

```
cout << “Neki text";
```

- On takođe radi sa predefinisanim manipulatorima (elementima koji menjaju formatiranje, npr. endl);

```
cout << “Neki text" << endl;
```

- U ovom primeru manipulator endl predstavlja znak za prelazak u novi red.

Formatiranje izlaza

- Za određivanje jednake širine izlaza, koristi se manipulator **setw** ili operacija **width**.

```
double values[] = { 1.23, 35.36, 653.7, 4358.24 };
for( int i = 0; i < 4; i++ )
{
   cout.width(10);
   cout << values[i] << '\n';
}
```

- U gornjem primeru se pri ispisivanju vrednosti svaki red dopunjava blanko znakovima do 10 znakova. Ukoliko želimo da se vrši popuna nekim drugim znakom, onda se koristi operacija **fill**.

```
for( int i = 0; i < 4; i++ )
{
   cout.width( 10 );
   cout.fill( '*' );
   cout << values[i] << endl
}
```

- Ukoliko se koristi **setw** sa argumentima, vrednosti se štampaju u poljima iste dužine. U tom slučaju se mora uključiti i IOMANIP.H datoteka.

```
double values[] = { 1.23, 35.36, 653.7, 4358.24 };
char *names[] = { "Zoot", "Jimmy", "Al", "Stan" };
for( int i = 0; i < 4; i++ )
cout << setw( 6 )  << names[i]<< setw( 10 ) <<
   values[i] << endl;
```

Ovim se dobijaju dve kolone, u jednoj su ispisana imena, a u drugoj vrednosti.

Operacije izlaznog toka

- Kao i kod ulaznog toka, i ovde se koriste iste operacije za otvaranje i zatvaranje toka (**open** i **close**).

- Za rad sa karakterima koristi se operacija **put**. Ona smešta zadati znak u izlazni tok. Na primer:

```
cout.put ('A');
```

- daje isti rezultat kao i

```
cout << 'A';
```

Funkcija write

- Funkcija **write** se koristi za upis bloka memorije u izlazni tok. Ona ima dva argumenta: prvi, **char** pokazivač i drugi, broj bajtova za upis. Treba napomenuti da je obavezna konverzija u **char*** pre adrese strukture ili objekta. Funkcija write se koristi i za binarne datoteke.
- Primer za upis strukture **Radnik** u binarnu datoteku:

```
Radnik r;
ofstream os("plata.dat", ios::binary);
os.write((char*) &r, sizeof(r));
os.close();
```

- Funkcije **seekp** i **tellp** su gotovo identične funkcijama **seekg** i **tellg**, jedino što se koriste za izlazne tokove.

Preklapanje operatora umetanja za korisničke tipove

- Preklapanje operatora umetanja (<<) za neku klasu dovodi do drugačijeg format ispisivanja u izlazni tok. Može se koristiti kao sredstvo za formatiranje izlaza. Koristi se zajedno sa manipulatorima.

```
#include <iostream.h>

class complex {
      double real, imag;
      friend ostream& operator<<(ostream&, complex);
…
}

ostream& operator<< (ostream &os, complex c){
      return os << “(“ <<c.real<< “,“<<c.imag<< “)“;
}

void main() {
   complex c(1.2, 2.3);
   cout << “c=“<<c;_
}
```

Primer

```
void main()
{
    Datum dt(4,11,2004);
    cout << dt;
}
```

```
class Datum
{
public:
    int dan, mesec, godina;
    Datum (int d, int m, int g){
        dan = d; mesec = m; godina = g;
    }
    friend ostream& operator<< ( ostream& os, Datum& dt );
};

ostream& operator<< ( ostream& os, Datum& dt )
{
    os << dt.dan << "." << dt.mesec << "." << dt.godina;
    return os;
}
```

Standardna biblioteka - STL

Standardna biblioteka STL

STL\Standard C++ Library Overview.htm

using namespace std

Kontejnerske klase

- Sequences
  - C++ Vectors
  - C++ Lists
  - C++ Double-Ended Queues
- Container Adapters
  - C++ Stacks
  - C++ Queues
  - C++ Priority Queues
- Associative Containers
  - C++ Bitsets
  - C++ Maps
  - C++ Multimaps
  - C++ Sets
  - C++ Multisets

Namespace

```
namespace X {
        int i,j,k;
}

int k;

void f1()
{
        int i=0;
        using namespace X;
        i++;   //lokalno i
        j++;   //X::j
        k++;   //greska X::k  ili globalno k?
        ::k++; //globalno k
        X::k++;       //k iz X
}
```

```
void  f2(){

int i=0;

using X::i;   //greska dva puta
deklarisano

using X::j;

using X::k;  //skriva globalno k

i++;

j++;

k++;}
```

Klase opste namene
<utility>
<functional>
<memory>
<ctime>

Iterator
<iterator>

Algoritmi
<algorithm>
<cstdlib>

Dijagnostika
<exception>
<stdexcept>
<cassert>
<cerrno>

Stringovi
<string>
<cctype>
<cwctype>
<cstring>
<cwchar>
<cstdlib>

Ulaz/Izlaz
<iostream>
<ios>
<streambuf>
…

Lokalizacija
<locale>
<clocale>

Support
<limits>
<climits>
<cfloat>
<new>
<typeinfo>
<ctime>
<csignal>
….

Numericki
<complex>
<valarray>
<numeric>
<cmath>

Vector

- Kontejnerska klasa
- definisana u std namespace
- #include <vector>

```
template<class T, class A = allocator<T> > class std::vector {
        public:
                typedef T value_type;
                typedef A allocator_type;
                typedef typename A::syze_type size_type;
                typedef typename A::difference_type difference_type;

                typedef implementation_dependent1 iterator;
                typedef implementation_dependent1 const_iterator;
                typedef std::reverse_iterator<iterator> reverse_iterator;
                typedef std::reverse_iterator<const iterator> const_reverse_iterator;

…
}
```

vector.h - STL\vectorHeader.txt

Primer

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

 vector<int> intVector;
 vector<int>::iterator vIterator;

  for( int i=0; i < 10; i++ )
    intVector.push_back(i);

  int total = 0;
  vIterator = intVector.begin();

  while( vIterator != intVector.end() ) {
    total += *vIterator;
    vIterator++;
  }
  cout << "Total=" << total << endl;
```

metode klase STL\C++ Vectors.htm
operatori klase STL\STL Vector Class.htm
back() – opis STL\back.htm

Iteratori

Klase ciji objekti predstavljaju pozicije elemenata u razlicitim kontejnerskim klasama

- Iteratori se uvek pridruzuju nekoj kontejnerskoj klasi
- Tip zavisi od kontejnerske klase kojoj su pridruzeni

```
#include <list>
#include <vector>
using namespace std;
...
list<int> nums;
list<int>::iterator nums_iter;
vector<double> values;
vector<double>::iterator values_iter;
vector<double>::const_iterator const_values_iter;
```

Tip Iteratora	Osobina
• Forward	Moze da specificira poziciju jednog elementa u kontejneru.
	Moze da menja vrednost od elementa do elementa u jednom smeru.
• Bidirectional	Isto kao i Forward ali moze da menja vrednost u oba smera
• Random access	Moze da se menja uoba smera u rupnijim koracima

Iterator:

- Konstantni - samo pregled sadrzaja bez promene
- Nekonstantni – i promena sadrzaja je omogucena

Operatori iteratora: STL\STL Iterator Classes.htm

```
#include <list>              // list class library
using namespace std;
...
  list<int> nums;
  list<int>::iterator nums_iter;
  nums.push_back (0);
  nums.push_back (4);
  nums.push_front (7);

  cout << endl << "List 'nums' now becomes:" << endl;
  for(nums_iter=nums.begin();nums_iter!=nums.end();nums_iter++)
  {
    *nums_iter = *nums_iter + 3; // Modify each element.
    cout << (*nums_iter) << endl;
  }
  cout << endl;
```

```
#include <list>     // list class library
#include <string> // string class library
using namespace std;
...
  list<string> words;
  list<string>::iterator words_iter;
...
  unsigned int total_length = 0;
  for (words_iter=words.begin(); words_iter != words.end();
       words_iter++)
  {
    total_length += (*words_iter).length();  // correct
//  total_length += *words_iter.length();    // incorrect !!
  }
  cout << "Total length is " << total_length << endl;
```

```
#include <vector>       // vector class library
#include <list>           // list class library
#include <algorithm>                // STL algorithms class library
using namespace std;
...
  vector<string> vec1;

  string state1 = "Wisconsin";
  string state2 = "Minnesota";
  vec1.push_back (state1);
  vec1.push_back (state2);
  vec1.push_back ("Illinois");
  vec1.push_back ("Michigan");
  sort(vec1.begin(),vec1.end()); // Sort the vector of strings.

  vec1.erase(vec1.begin(),vec1.end());
...
  list<int> nums;
  list<int>::iterator nums_iter;
...
  nums.erase (nums.begin(), nums.end()); // Remove all elements.
...
  nums_iter = find(nums.begin(), nums.end(), 3); // Search the list.
  if (nums_iter != nums.end())
  {
    cout << "Number " << (*nums_iter) << " found." << endl;
  }
...
  // If we found the element, erase it from the list.
  if (nums_iter != nums.end()) nums.erase(nums_iter);
```

Lists

Liste su sekvence elemenata smestene u povezane liste

Container constructors	create lists and initialize them with some data
Container operators	assign and compare lists
assign	assign elements to a list
back	returns a reference to last element of a list
begin	returns an iterator to the beginning of the list
clear	removes all elements from the list
empty	true if the list has no elements
end	returns an iterator just past the last element of a list
erase	removes elements from a list
front	returns a reference to the first element of a list
insert	inserts elements into the list
max_size	returns the maximum number of elements that the list can hold
merge	merge two lists

pop_back	removes the last element of a list
pop_front	removes the first element of the list
push_back	add an element to the end of the list
push_front	add an element to the front of the list
rbegin	returns a reverse_iterator to the end of the list
remove	removes elements from a list
remove_if	removes elements conditionally
rend	returns a reverse_iterator to the beginning of the list
resize	change the size of the list
reverse	reverse the list
size	returns the number of items in the list
sort	sorts a list into ascending order
splice	merge two lists in constant time
swap	swap the contents of this list with another
unique	removes consecutive duplicate elements

Algoritmi

- u <algorithm> STL\C++ Algorithms.htm

Primer za sort()

```
#include <vector>
#include <algorithm>  // Include algorithms
using namespace std;

vector<int> vec;
vec.push_back (10);
vec.push_back (3);
vec.push_back (7);

sort(vec.begin(), vec.end()); // Sort the vector

// The vector now contains: 3, 7, 10
```

Primer: find( )

```
#include <vector>  // vector class library
#include <list>          // list class library
#include <algorithm>     // STL algorithms class library
using namespace std;
...
list<int> nums;
list<int>::iterator nums_iter;

nums.push_back (3);
nums.push_back (7);
nums.push_front (10);

nums_iter = find(nums.begin(), nums.end(), 3); // Search the list.
if (nums_iter != nums.end()) {
    cout << "Number " << (*nums_iter) << " found." << endl; // 3
}
else {
    cout << "Number not found." << endl;
}

// If we found the element, erase it from the list.
if (nums_iter != nums.end()) nums.erase(nums_iter);
// List now contains: 10 7
```

String

- #include <string>

- STL\ANSI String Class.htm