OBJEKTNO PROGRAMIRANJE 1

Oznaka predmeta: OOP
Predavanje broj: 12

Nastavna jedinica: Nasleđivanje. Templejti i izuzeci.

Nastavne teme:

Virtuelni destruktor. Nizovi i nasleđivanje. Apstraktne klase. Generički mehanizam. Šabloni klasa. Nasleđivanje i generisanje. Generisanje funkcija. Razrešavanje poziva funkcija. Šabloni funkcija članica. Deklaracije i definicije šablona. Obrada izuzetaka. Postavljanje i hvatanje izuzetka. Mehanizam obrade izuzetaka u jeziku C++. Konstruktori i destruktori. Specifikacije izuzetaka. Specijalne funkcije za obradu izuzetaka: terminate, unexpected. Lambda.

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

Literatura:

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

Virtuelni destruktor

```
#include <iostream>
using namespace std;
class Base{ public:
    Base()          {cout<<"[Base] konstruktor \n";}
    virtual ~Base() {cout<<"[Base] destruktor  \n";}
};
class Derived:public Base{ public:
    Derived()  {cout<<"[Derived] konstruktor \n";}
    ~Derived() {cout<<"[Derived] destruktor  \n";}
};
int main(){
    Base* pbase = new Derived;
    delete pbase; system("pause"); return 0;
}
```

Konstruktor se poziva pre nego što se objekat kreira, pa nema smisla da konstruktor bude virtuelan

- Virtuelni destruktor obezbeđuje da se pri pozivu delete *pbase* izvrši destruktor *~Derived*() za objekat klase *Derived*.
- Ako destruktor osnovne klase ne bi bio virtuelni poziv destruktora bi se rešio statičkim vezivanjem, prema tipu pokazivača, pa bi se pozvao destruktor *~Base*() a radi se o objektu klase *Derived*.
  - Preporuka: Ako neka osnovna klasa ima neku virtuelnu funkciju, trebalo bi da i njen destruktor (ako ga ima) bude virtuelan.
- Unutar virtuelnog destruktora izvedene klase automatski se poziva destruktor osnovne klase.

Nizovi i nasleđivanje

- Niz objekata izvedene klase nije jedna vrsta niza objekata osnovne klase. Radi se o tome da je objekat izvedene klase veći ("duži"). Evo gde je problem:

```
class Base                      { public: int bi; };
class Derived : public Base     { public: int di; };
void f(Base *b) { cout<<b[2].bi; }
void main () {
  Derived d[5];
  d[2].bi=77;
  f(d);           // Derived[] se konvertuje u Derived*,
}                 // a zatim u Base*; neće se ispisati 77!
//funkcija f smatra da radi nad nizom objekata osnovne klase, jer ima
//formalni argument tipa Base*.
```

- Objekti osnovne klase su "kraći" (nemaju sve članove) od objekata izvedene klase.
  - Kada se funkciji f prosledi niz objekata izvedene klase (koji su duži), preko pokazivača tipa Derived* (koji se implicitno konvertuje u Base*), funkcija nema načina da odredi da se niz sastoji samo od objekata izvedene klase.
  - Izračunavanje pomeraja za pristup određenom elementu niza unutar funkcije f zasniva se na veličini objekta tipa Base. Rezultat je zato, u opštem slučaju, neodređen.

Nizovi i nasleđivanje

- Nije dobro u niz objekata osnovne klase smeštati direktno (eksplicitnom konverzijom) objekte izvedene klase, jer su oni "duži", a za svaki element niza je odvojen samo prostor koji je dovoljan za smeštanje objekta osnovne klase.
- Kolekcije objekata treba realizovati kao kolekcije pokazivača na objekte:

```
void f(Base **b, int i) { cout<<b[i]->bi; }
void main () {
  Derived d0,d1; Base b2;
  Base *b[3];                          // b je tipa Base*[];
  b[0]=&d0; b[1]=&d1; b[2]=&b2;        // Derived* → Base*;
  d1.bi=77;
  f(b,1);                              // ispisaće se 77;
}
```

- Kako je objekat izvedene klase jedna vrsta objekta osnovne klase, dozvoljena je implicitna konverzija pokazivača Derived* u Base*.
- Nije dozvoljena implicitna konverzija pokazivača Derived** u Base**.

```
void main () {
  Derived *d[5]; // d se može konvertovati u tip Derived**;
  f(d,2);        // nije dozvoljena konverzija Derived** u Base**
}
```

- Kolekcije objekata korisničkih tipova treba realizovati kao kolekcije pokazivača.

Apstraktne klase

- Čest je slučaj da osnovna klasa nema niti jedan konkretan primerak (objekat), već samo predstavlja generalizaciju izvedenih klasa (npr. osnovna je "osoba" a izvedene klase su "muškarac" i "žena").
- Klasa koja nema instance (objekte), već služi samo da se iz nje izvode druge klase, naziva se *apstraktnom klasom* (*abstract class*).
- Generalizacija (*generalization*) se primenjuje kada se zajedničke osobine nekoliko specifičnih klasa grupišu u zajedničku osnovnu (apstraktnu) klasu, samo u cilju postizanja polimorfizma.
- *Specijalizacijom* (*specialization*) se iz neke konkretne klase, koja ima svoje objekte, izvodi druga klasa, koja predstavlja objekte posebne vrste.
- Apstraktna klasa sadrži bar jednu virtuelnu funkciju članicu koja je u njoj samo deklarisana.
  - definicije te virtuelne funkcije daće izvedene klase.
  - ovakva virtuelna funkcija naziva se *čistom virtuelnom funkcijom* (*pure virtual function*).
    Njena deklaracija u osnovnoj klasi završava se sa specifikatorom =0:

```
class OutCharDevice {  //...
  virtual OutStatus put (char) = 0;  // čista virtuelna funkcija
};
```

Virtuelne funkcije – primer 1/4

```
//_____________________fajl ClanBiblioteke.h_____________________
#pragma once
#include <iostream>
#include <string>
using namespace std;
#pragma once
class ClanBiblioteke {
   protected:
        static double clanarina;
        string imeclana;
        double racun;
   public:
                        ClanBiblioteke(string imeclana, double racun);
        virtual double plati_clanarinu (); // virtuelna funkcija
                string get_imeclana();
        virtual         ~ClanBiblioteke();  // vitruelni destruktor
};
//_____________________fajl ClanBiblioteke.cpp_____________________
#include "ClanBiblioteke.h"
ClanBiblioteke::ClanBiblioteke(string imeclana, double racun){
   this->imeclana = imeclana;
   this->racun    = racun;
```

Virtuelne funkcije – primer 2/4

```
  cout<<"ClanBiblioteke"<<endl;
}
double ClanBiblioteke::clanarina = 10.00;
double ClanBiblioteke::plati_clanarinu(){ return racun-=clanarina;     }
string ClanBiblioteke::get_imeclana()   { return imeclana;             }
       ClanBiblioteke::~ClanBiblioteke(){cout<<"~ClanBiblioteke"<<endl;}
//_____________________fajl PocasniClanBiblioteke.h____________________
#pragma once
#include "ClanBiblioteke.h"
class PocasniClanBiblioteke : public ClanBiblioteke {
  public:
            PocasniClanBiblioteke(string imeclana, double racun);
     double plati_clanarinu ();
            ~PocasniClanBiblioteke();
};
//____________________fajl PocasniClanBiblioteke.cpp___________________
#include "PocasniClanBiblioteke.h"
PocasniClanBiblioteke::PocasniClanBiblioteke(string imeclana,
                                             double racun)
: ClanBiblioteke(imeclana,racun)
{ cout<<"PocasniClanBiblioteke"<<endl; }
```

Virtuelne funkcije – primer 3/4

```
double PocasniClanBiblioteke::plati_clanarinu () {
  return racun; //besplatna clanarina
}
PocasniClanBiblioteke::~PocasniClanBiblioteke(){
  cout<<"~PocasniClanBiblioteke"<<endl;
}
//_____________________________fajl gde je main_____________________________
#include "ClanBiblioteke.h"
#include "PocasniClanBiblioteke.h"
using namespace std;
const int BROJ_CLANOVA=2;
int main(void){
   ClanBiblioteke* clanovi[BROJ_CLANOVA];
   cout<<endl; clanovi[0] = new ClanBiblioteke("Mirko", 100.00);
   cout<<endl; clanovi[1] = new PocasniClanBiblioteke("Slavko", 100.00);
   cout<<endl;
   for (int i=0; i<BROJ_CLANOVA; i++)
     cout<<clanovi[i]->get_imeclana()
         <<" ima nakon placanja clanarine na racunu "
         <<clanovi[i]->plati_clanarinu()<<" din."<<endl;
   for (int i=0; i<BROJ_CLANOVA; i++) { cout<<endl; delete clanovi[i]; }
   cout<<endl; system("pause"); return 0;
}
```

Virtuelne funkcije – primer 4/4

```
//IZLAZ

ClanBiblioteke

ClanBiblioteke
PocasniClanBiblioteke

Mirko ima nakon placanja clanarine na racunu 90 din.
Slavko ima nakon placanja clanarine na racunu 100 din.

~ClanBiblioteke

~PocasniClanBiblioteke
~ClanBiblioteke

Press any key to continue . . .
```

Šabloni klasa

- Šablonom klase definiše se izgled cele familije klasa koje imaju isti izgled.
- Šablon klase ponekad se naziva i *generičkom klasom* (*generic class*), a konkretna klasa generisana iz šablona u terminologiji jezika C++ naziva se *template class*.
- Generička klasa (šablon) za matrice elemenata raznih tipova:

```
template <class T>    // ili template <typename T>
  class matrix {
      T **m;
      int columns, rows;  // broj kolona i vrsta matrice;
      //...
    public:
      matrix (int col, int row);
      //...
  };
```

- Deklaracija šablona u potpunosti odgovara deklaraciji obične klase, samo što ispred nje stoji template čiji je formalni argument tip T naveden između znakova < i >.
- Tamo gde se u deklaraciji šablona pojavljuje ovaj identifikator T, pri generisanju konkretne klase, sa konkretnim tipom X kao stvarnim argumentom, stajaće tip X. Identifikator matrix je ime šablona.

Šabloni klasa

- Argumenti šablona mogu biti i objekti proizvoljnog tipa, koji će pri generisanju klase biti zamenjeni konkretnim vrednostima.
  Šablon klase vector, realizuje statički niz elemenata tipa T dimenzije N koja se specificira pri generisanju klase.

```
template <class T, int N>    //N se specificira pri generisanju klase
  class vector {
    T v[N];
  public:
    T& operator[](int i) {
      if ((i>=0) && (i<N)) return v[i];
      else error("Boundary check error!");
    }
  };
```

- Konkretna klasa se specificira kada se negde u programu upotrebi ime šablona sa listom stvarnih argumenata šablona. Na primer:

```
matrix<int>       m1(7,3);
matrix<complex>   m3(4,4);
vector<complex,5> v;
typedef vector<complex,100> cvec100; //sinonim za vector<complex,100>.
```

- Izraz matrix<int> je ime klase koja se dobija zamenom tipa int na mestima gde se pojavljuje tip T u šablonu matrix.

Šabloni klasa

- Ako se negde u programu specificira klasa koja je dobijena konkretizacijom šablona, onda će prevodilac generisati stvarni izgled te klase, zamenom formalnih argumenata stvarnim argumentima šablona.
- Prevodilac te klase generiše interno, tako da je efekat isti kao da je programer sâm ispisao sve konkretne klase, sa različitim identifikatorima.
- Identifikator konkretizovane (generisane) klase sastoji se od imena šablona i liste stvarnih argumenata šablona između znakova < i >.
- Ime generisane klase je potpuno ravnopravno sa imenima običnih klasa.
- Tipovi stvarnih argumenata u imenu generisane klase moraju se u potpunosti poklapati sa tipovima formalnih argumenata šablona.
- Stvarni argumenti šablona koji ne predstavljaju tipove (čiji formalni argumenti nisu class) moraju biti konstantni izrazi, adrese objekata ili funkcija sa eksternim povezivanjem, ili adrese statičkih članova klasa. Oni se u potpunosti izračunavaju u fazi prevođenja.
- Dva imena generisane klase predstavljaju isti tip, samo ako imaju isto ime šablona, i ako im stvarni argumenti imaju u potpunosti identične vrednosti.

```
vector<complex,5  > v1;
vector<complex,2+3> v2;  //objekti v1 i v2 su istog tipa
```

Nasleđivanje i generisanje

- Skica za kolekciju koja predstavlja listu je:

```
class Object {
  friend class List;  Object *next;
};
class List {
  Object *head;  // glava liste;
  //...
public:
  void put (Object*);  Object* get();  //...
};
class jabuka    : public Object {/*...*/};
class automobil : public Object {/*...*/};
void main () {
  jabuka j;   automobil a;
  List ListaJabuka;      //liste elemenata raznih tipova
  List ListaAutomobila;  //kao objekti jedne iste klase List
  ListaJabuka.put(&j);
  ListaAutomobila.put(&a);
}
```

- Ovde razni korisnički tipovi, koji inače mogu da modeluju sasvim različite i nezavisne entitete, su naslednici istog osnovnog tipa Object. Time se nasilno proizvodi činjenica da potpuno nezavisni tipovi imaju zajednička svojstva.

Nasleđivanje i generisanje

- U prethodnom primeru, objekti tipa jabuka i tipa auto podvedeni su pod zajednički tip Object, iako nemaju nikakve međusobne veze.
  - U datom primeru može se dogoditi sledeće:

```
ListaJabuka.put(&a);
ListaJabuka=ListaAutomobila;
```

    - kolekcija objekata izvedene klase nije jedna vrsta kolekcije objekata osnovne klase.
- Za realizaciju kolekcije bilo je neophodno da svi tipovi budu naslednici jednog apstraktnog tipa.
- Generički mehanizam rešava navedeni problem.
- Šablonima se definiše zajednički *izgled* klasa koje mogu da modeluju sasvim različite entitete.
  - Generisane klase se ne podvode ni pod kakav zajednički tip, već predstavljaju sasvim nezavisne tipove (List<jabuka> i List<automobil>).
  - Šabloni se najčešće koriste za realizaciju kontejnerskih klasa.
- Šabloni predstavljaju notacionu pogodnost za jedinstveno opisivanje skupa klasa koje imaju istu *realizaciju*, ali modeluju nezavisne entitete.
- Nasleđivanje modeluje relacije između klasa koje imaju zajedničke *osobine* a različite realizacije.

Generisanje funkcija

- Neka je potrebna familija funkcija koje realizuju sortiranje niza elemenata raznih tipova T. Familija funkcija koje imaju isti izgled, samo operišu sa različitim tipovima, definiše se šablonom funkcije.
- Šablon funkcije naziva se *generičkom funkcijom* (*generic function*), a konkretna funkcija generisana iz šablona, kao i obična funkcija koja po tipovima svojih argumenata odgovara tipu šablona zove se *template function*. Na primer:

```
template <class T>
  void sort (T *niz, int n) {
    for (int i=0; i<n-1; i++)
      for (int j=i+1; j<n; j++)
        if (niz[i]>niz[j]) swap(niz[i],niz[j]); // zamena mesta
  }
```

- Kada se generiše konkretna funkcija po šablonu sort, svugde gde se u šablonu pojavljuje ime tipa T, biće zamenjen tip X koji je dostavljen kao stvarni argument.
  - Za ovaj tip X mora postojati definisan **operator >** i funkcija swap, jer će se oni pozivati iz tela generisane funkcije.
- U deklaraciji šablona za funkciju može se pojaviti i specifikacija generisane funkcije ili klase, koja kao svoj argument ima formalni argument nekog šablona. `template <class T> void sort (vector<T>&);`

Generisanje funkcija

- Za generisanje odgovarajuće verzije funkcije, potrebna je definicija šablona funkcije.
- Za generisanje poziva funkcije, potrebna je samo deklaracija šablona funkcije, poput prethodno navedene deklaracije.
- Svaki formalni argument šablona MORA biti upotrebljen u deklaracijama tipova formalnih argumenata funkcije. Sledeće je pogrešno:

```
template <class T> T create();  // greška!
```

- Šablonom funkcije specificira se familija proizvoljnog broja preklopljenih funkcija.
- Primenjuju se pravila za razrešavanje poziva preklopljenih funkcija.

```
void f (vector<complex>& cv, vector<int>& iv) {
  sort(cv);  // poziva se sort(vector<complex>);
  sort(iv);  // poziva se sort(vector<int>);
}
```

- Svako pojavljivanje imena funkcije koje je deklarisano u šablonu, podvrgava se pravilima za razrešavanje preklopljenih funkcija.
- Funkcija definisana šablonom može biti preklopljena bilo sa ostalim običnim funkcijama sa istim imenom, bilo sa drugim funkcijama istog šablona.

Razrešavanje poziva funkcija

1. - Najpre se traži funkcija koja obezbeđuje potpuno poklapanje tipova stvarnih argumenata sa tipovima formalnih argumenata (ako se nađe poziva se).

2. - Traži se šablon funkcije kojim se može generisati funkcija sa potpunim slaganjem formalnih argumenata (bez trivijalnih konverzija); ako se nađe, poziva se ta generisana funkcija.

3. - Primenjuju se uobičajena pravila za razrešavanje poziva, ako se nađe odgovarajuća funkcija, poziva se.

•Poziv je pogrešan ako se ne nađe odgovarajuća funkcija ili je poziv dvosmislen.

•Konkretna funkcija ne mora samo da se generiše pomoću poziva, već i eksplicitnom deklaracijom. Takvo rešenje omogućuje da se generiše odgovarajuća funkcija, i da se kasnije ona poziva uz uobičajena pravila za konverziju. Na primer:

```
template<class T> T maxx(T a,T b) { return a>b ? a : b;}
//int maxx(int,int); // ako se doda ova deklaracija nece biti greske


void f(int a, int b, char c, char d) {
  int  m1=maxx(a,b);  // generiše se    int  maxx(int,int);
  char m2=maxx(c,d);  // generiše se    char maxx(char,char);
  int  m3=maxx(a,c);  // greška: ne može se generisati maxx(int,char)!
}
```

Šabloni funkcija članica

- Funkcija članica generičke klase (šablona klase) je implicitno i sama generička funkcija, sa argumentima šablona klase kao argumentima šablona funkcije.
- Za svaku konkretnu generisanu klasu, generišu se i sve njene konkretne funkcije članice.

```
template <class T>
  class matrix {
    T **m;
    int columns, rows;  // broj kolona i vrsta matrice;
    //...
  public:
    //...
    matrix (int row, int col);
    T& operator()(int row, int col);  // operator();
    //...
  };
```

- Funkcija članica operator() može biti definisana na sledeći način:

```
template<class T>
  T& matrix<T>::operator() (int row, int col) {
    if ((col>=0 && col<columns) &&
        (row>=0 && row<rows) ) return m[row][col];
    else error("Boundary check error!");
  }
```

Šabloni funkcija članica

- U imenu konstruktora šablona klase ne pojavljuju se argumenti šablona klase (oni su implicitni).

```
template<class T>
  matrix<T>::matrix<T> (int row, int col) { // greška!
    //...
  }

template<class T>
  matrix<T>::matrix (int row, int col) { // u redu;
    //...
  }
```

- Funkcije članice se pozivaju na sledeći način:

```
matrix<int>     mi(5,5);
matrix<complex> mc(4,4);
mi(1,1)=0;                          //poziva se matrix<int>::operator()
mc(2,2)=complex(0,0);          //poziva se matrix<complex>::operator()
matrix<int>& rmi=matrix<int>(5,5);// poziv konstruktora
mi.~matrix();                       // poziv destruktora
```

Šabloni funkcija članica

- Za razliku od funkcija članica, prijateljske funkcije generičke klase nisu implicitno generičke funkcije. Na primer:

```
template<class T>
  class List {
    friend void count ();
    friend T max(List<T>);
    friend void f(List);  // greška! Mora konkretno npr. List<int>
    //...
  };
```

- Funkcija count nije generička, i postoji samo jedna ova funkcija koja je prijatelj svih klasa generisanih iz ovog šablona.
- Funkcija max je generička, i svaka klasa generisana iz šablona imaće svoju prijateljsku funkciju max odgovarajućeg tipa.
- Deklaracija funkcije f predstavlja grešku, jer ne postoji tip List, već samo konkretizovani tipovi List<int>, List<complex> itd.
- Svaka klasa, odnosno funkcija generisana iz šablona, poseduje svoje sopstvene statičke članove, odnosno statičke lokalne objekte.

Deklaracije i definicije šablona

- Deklaracija šablona javlja se samo kao globalna deklaracija.
- U programu može postojati samo jedna definicija svakog šablona, a dozvoljeno je postojanje više deklaracija istog šablona.
  - Svaka upotreba imena šablona klase predstavlja deklaraciju konkretne generisane klase.
  - Poziv generičke funkcije ili uzimanje njene adrese predstavlja deklaraciju konkretne generisane funkcije.
- Ako se definiše obična funkcija sa istim imenom i sa potpuno odgovarajućim tipovima kao što je šablon funkcije, onda ova definicija predstavlja definiciju konkretne generisane funkcije za date tipove.

```
template<class T>
  void sort(vector<T>& v) {/*...*/}  // za sve ostale pozive

void sort(vector<int>& v) {/*.../*}   // za pozive vector<int>
```

- Konkretna definicija funkcije
  `sort(vector<int>&)`
  biće upotrebljena za pozive sa odgovarajućim tipom argumenata `vector<int>` dok će se funkcije za ostale tipove argumenata generisati iz šablona.
  Isto važi i za šablone klasa.

Mali primer

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};

// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) ;
    char increase ();
};
```

Mali primer

```
mycontainer<char>::mycontainer(char arg){
  element=arg;
}

char mycontainer<char>::increase()
{
    if ((element>='a')&&(element<='z'))
      element+='A'-'a';
    return element;
}

int main ()
{
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.increase() << endl;
  system("pause");
  return 0;
}
```

Primer sa stack-om 1/4

```
#pragma once
template <class T>
class MyStack{
  private:
    T* stackPtr ;
    int size ;  // Number of elements on Stack
    int index ;
    static const int max_stack = 1000;
  public:
    MyStack(int = 10) ;
    ~MyStack() { delete [] stackPtr ; }
    bool push(const T&);
    bool pop(T&) ;  // pop an element off the stack
    bool isEmpty() const { return index == -1 ; }
    bool isFull() const  { return index == size - 1 ; }
} ;
//constructor with the default size 10
template <class T>
MyStack<T>::MyStack(int s){
    size = (s > 0 && s < max_stack) ? s : 10 ;
    index = -1 ;  // initialize stack
    stackPtr = new T[size] ;
}
```

Primer sa stack-om 2/4

```
// push an element onto the Stack
template <class T>
bool MyStack<T>::push(const T& item)
{
  if (!isFull())
  {
    stackPtr[++index] = item ;
    return true ;  // push successful
  }
  return false ;  // push unsuccessful
}
// pop an element off the Stack
template <class T>
bool MyStack<T>::pop(T& popValue)
{
  if (!isEmpty())
  {
    popValue = stackPtr[index--] ;
    return true ;  // pop successful
  }
  return false ;  // pop unsuccessful
}
```

Primer sa stack-om 3/4

```
#include <iostream>
#include "MyStack.h"
using namespace std ;
void main(){
  typedef MyStack<double> DoubleStack ; // for double members
  typedef MyStack<int> IntStack ;
  DoubleStack ds(5) ;
  double f = 1.1 ;//must be a double type because of type od memebers
  cout << "Pushing elements onto fs" << endl ;
  while (ds.push(f)){ cout << f << ' ' ; f += 1.1 ; }
  cout << endl << "Stack Full." << endl
       << endl << "Popping elements from fs" << endl ;
  while (ds.pop(f)) cout << f << ' ' ;
  cout << endl << "Stack Empty" << endl ; cout << endl ;
  IntStack is ;
  int i = 1 ;
  cout << "Pushing elements onto is" << endl ;
  while (is.push(i)){ cout << i << ' ' ; i += 1 ;}
  cout << endl << "Stack Full" << endl
       << endl << "Popping elements from is" << endl ;
  while (is.pop(i)) cout << i << ' ' ;
  cout << endl << "Stack Empty" << endl ;system("pause");}
```

Primer sa stack-om 4/4

- Izlaz:

```
Pushing elements onto fs
1.1 2.2 3.3 4.4 5.5
Stack Full.

Popping elements from fs
5.5 4.4 3.3 2.2 1.1
Stack Empty

Pushing elements onto is
1 2 3 4 5 6 7 8 9 10
Stack Full

Popping elements from is
10 9 8 7 6 5 4 3 2 1
Stack Empty
Press any key to continue . . .
```

Obrada izuzetaka

- Dobar mehanizam obrade izuzetaka trebalo bi da realizuje sledeće:
  - potrebno je da se informacija o izuzetku prenosi nezavisno od mehanizma prenosa argumenata i vraćanja vrednosti funkcije.
  - dobro je da ta informacija bude bilo kog apstraktnog tipa, koga može definisati korisnik (objekat neke klase).
  - struktura programa na mestu na kome se želi obrada izuzetka treba da bude jasna, tako da su precizno izdvojena dva dela:
    - jedan koji predstavlja osnovni tok programa,
    - i drugi koji obuhvata kôd za obradu izuzetka.

- Deo koji predstavlja osnovni tok programa ne treba da bude opterećen ispitivanjem postojanja izuzetka, već tok programa treba da automatski pređe na deo za obradu izuzetka, ako se izuzetak pojavi.

- U telu funkcija u kojima ne nastaju izuzeci, niti se oni obrađuju, ne treba da postoji nikakav dodatni kôd vezan za prosleđivanje izuzetka iz pozvane u pozivajuću funkciju.

Postavljanje i hvatanje izuzetka

- Ovaj mehanizam omogućava prenos kontrole toka programa i informacije sa mesta u programu gde nastaje izuzetak, na mesto u hijerarhiji poziva na kome postoji spremnost da se taj izuzetak obradi.
- Ovo mesto na kome se obrađuje izuzetak nije poznato onome ko izuzetak postavlja.

Mehanizam obrade izuzetaka u jeziku C++

- Na mestu gde nastaje izuzetna situacija *postavlja* se izuzetak.
- Informacija koja opisuje izuzetak može biti objekat proizvoljnog tipa.
- Izuzetak se postavlja izrazom throw.

```
void Table::put (T t) {
  T *pt=new T(t);
  if (!pt)
    throw ERR; //postavljanje izuzetka npr. tipa Table::Status;
  //...
}
```

- Izraz throw uzrokuje da se kontrola toka programa prebaci na ono mesto u hijerarhiji poziva funkcija, na kome postoji kôd koji obrađuje taj izuzetak.
- Kôd koji obrađuje izuzetak (*exception handler*) specificira se na sledeći način:

Mehanizam obrade izuzetaka u jeziku C++

```
void f() {  // funkcija pozivalac;
  Table Tbl;
  T t1,t2,t3;

  try {
    //...
    Tbl.put(t1);
    //... osnovni tok programa;
    Tbl.put(t2);
    //... osnovni tok programa;
    Tbl.put(t3);
    //... osnovni tok programa;
  }
  catch (Table::Status s) {
    //... obrada izuzetka izazvanog sa "throw ERR";
  }
}
```

- Kod koji specificira osnovni tok programa navodi se kao složena naredba (blok) iza ključne reči try.
- Svaki catch iskaz predstavlja kôd za obradu izuzetka (*exception handler*) koji je predstavljen objektom odgovarajućeg tipa.

Mehanizam obrade izuzetaka u jeziku C++

- Unutar iskaza catch deklarisan je lokalni objekat s tipa Table::Status, što označava da ovaj blok može da obradi izuzetak koji je postavljen pomoću izraza throw sa objektom istog tipa Table::Status.
- Iza reči catch sledi deklaracija formalnog argumenta bloka za obradu izuzetka.
  - Ovaj argument predstavlja objekat koji se prihvata kada se generiše odgovarajući izuzetak.
  - Iza zagrada sledi složena naredba (blok), koja predstavlja kôd za obradu izuzetka.

- Izvršavanjem izraza throw iza koga je navedena vrednost nekog tipa kontrola toka programa se prebacuje na najbliže mesto u hijerarhiji poziva, koje poseduje catch iskaz sa argumentom odgovarajućeg tipa.
- Preko ove vrednosti se informacija o izuzetku prenosi sa mesta nastanka na mesto obrade.

Mehanizam obrade izuzetaka u jeziku C++

- Kôd za obradu izuzetka (*handler*) će biti izvršen samo od strane throw izraza unutar istog try bloka, ili od strane throw izraza u funkcijama koje se pozivaju (direktno ili indirektno) unutar istog try bloka.
- Izuzetak se može obraditi u istom bloku u kome i nastaje.
- Sledi primer gde funkcija g() generiše izuzetak ali je obrada smeštena u pozivajućoj funkciji f():

```
class gException {/*...*/};

void g() {  //...
  if (/*...*/)
    throw gException();
  //...
}
void f () {
  try {
     g();
     throw "Izuzetak 2";
  }
  catch (char *p) {
    cout<<p;   // Obrađuje se izuzetak iz iste funkcije;
  }
  catch (gException& gE) {
    cout<<"Izuzetak u funkciji g!";
  }
}
```

Mehanizam obrade izuzetaka u jeziku C++

- Unutar catch bloka ne mora se uopšte koristiti informacija o izuzetku; dovoljno je poznavanje činjenice da je izuzetak odgovarajućeg tipa nastao.
- U ovom primeru, tok izvršavanja funkcije f je sledeći.
  - Izvršava se složena naredba iza reči try.
  - Unutar ove složene naredbe poziva se funkcija g i ako se unutar funkcije g postavi izuzetak sa throw gException(), kontrola će se prebaciti na catch iskaz koji prihvata reference na objekat tipa gException generisan konstruktorom u iskazu throw. Inače, ako ovaj izuzetak ne postoji, funkcija g se normalno završava, i nastavlja se izvršavanje funkcije f.
  - U daljem toku funkcije f postavlja se izuzetak sa throw "Izuzetak 2", pa se kontrola prebacuje na iskaz catch koji prihvata objekat tipa char*.
  - Ako nijedan izuzetak u try bloku ne postoji, ovaj blok se završava, naravno bez izvršavanja catch iskaza.
- Iskaz try zajedno sa složenom naredbom iza je naredba.
- Iza reči throw može da stoji proizvoljni izraz.
- Postavljanje izuzetka izaziva prenos kontrole toka programa na najbliže mesto u hijerarhiji poziva na kome se taj izuzetak može obraditi. Na to mesto se prenosi objekat koji je rezultat izraza iza reči throw. Tip ovog objekta određuje koji iskaz catch može da obradi taj izuzetak.

Mehanizam obrade izuzetaka u jeziku C++

```
class DivisionError {
  double d;
public:
  DivisionError(double x) : d(x) {}
  void Handle() {cout<<endl<<"Division of "<<d<<" by zero!“<<endl;}
};
void f(double x, double y) {
  //...
  if (y==0) throw DivisionError(x);
  else z=x/y;
  //...
}
void main () {
  try { f(5,0); }
  catch (DivisionError& de) { de.Handle(); }
}
```

- Izrazom throw kreira se jedan privremeni objekat koji se smešta u statičku memoriju.
- Životni vek ovog objekta je ograničen, taj objekat je neimenovan, a smešta se u posebnu statičku oblast memorije, kako bi bio dostupan i na mestu na kome se izuzetak obrađuje.
- Ovaj privremeni objekat inicijalizuje se rezultatom izraza iza reči throw.
- Ovim privremenim objektom se inicijalizuje argument catch iskaza na mestu obrade izuzetka.

Mehanizam obrade izuzetaka u jeziku C++

- U prethodnom primeru, kreiraće se objekat koji će se inicijalizovati konstruktorom DivisionError(x) na mestu postavljanja izuzetka, a na mestu obrade će se inicijalizovati referenca ***de*** tako da upućuje na taj objekat.
- Ovaj objekat živeće sve dok živi i referenca.
- Treba primetiti da ovako kreirani objekat ne može biti automatski, jer referenca ***de*** u funkciji main ne može upućivati na automatski objekat funkcije g iz koje se izašlo u trenutku postavljanja izuzetka.
    - Automatski objekti se smeštaju na stek, koji se u trenutku postavljanja izuzetka vraća na stanje koje odgovara mestu obrade (pozivaoca).

- Osim u pogledu provere tipova i u pogledu opisanog korišćenja privremenog objekta, prenos objekta pri postavljanju i obradi izuzetka u potpunosti odgovara mehanizmu prenosa argumenata u funkciju ili vraćanju vrednosti iz funkcije.
- Prevodilac može optimizovati proces prenošenja izuzetka sa mesta postavljanja na mesto obrade.

Mehanizam obrade izuzetaka u jeziku C++

- Često je potrebno postojeći izuzetak samo delimično obraditi, a zatim ga proslediti višem nivou u hijerarhiji poziva.
- To je moguće obaviti izrazom throw bez argumenta.
- Ovakav izraz postavlja isti izuzetak koji se trenutno obrađuje.
- Ovakav izraz se može pojaviti samo unutar koda za obradu izuzetka, ili unutar koda funkcija koje se (direktno ili indirektno) pozivaju iz tog koda. Na primer:

```
try {
  //...
}
catch (...) {  // prihvata sve tipove izuzetaka;
  // .. obradi izuzetak delimično,
  throw; // i ponovo ga postavi, kako bi ga viši nivoi
         // obradili;
}
```

Obrada izuzetka

- Ako je u deklaraciji argumenta catch iskaza naveden tip T, const T, T& ili const T&, onda catch iskaz može obraditi izuzetke tipa E, ako vredi nešto od:
  - 1) T i E su isti tipovi;
  - 2) T je osnovna klasa klase E, dostupna na mestu postavljanja izuzetka,
  - 3) T je tip pokazivača, a E je tip pokazivača koji se može konvertovati u tip T standardnom konverzijom pokazivača, na mestu postavljanja izuzetka. (podržano nasleđivanje i polimorfizam, prenos i konverzija pokazivača i referenci na izvedene klase u pokazivače i reference na osnovne klase, dinamičko vezivanje)

```
class DeviceError {   //...
public:
 virtual void Handle();    //sta da je virtual void Handle() = 0;
  //...
};
class DiskError    : public DeviceError {/*...*/};
void f() {
  try {   //...  }
  catch (DeviceError &de) { // prihvati bilo koji tip DeviceError;
    de.Handle();  // dinamičko vezivanje;
  }
}
```

Obrada izuzetka

- Argument catch iskaza u primeru na prethodnom slajdu je referenca, a ne običan objekat, jer bi u tom slučaju bio prenesen samo objekat osnovne klase, i ne bi bio aktiviran virtuelni mehanizam. U ovom primeru, jedan catch iskaz prihvata sve greške apstraktnog tipa DeviceError i obrađuje ih korišćenjem polimorfizma.
- Iz liste catch iskaza jednog try bloka bira se onaj koji po tipu svog argumenta odgovara tipu izuzetka.
  - ako postoji više catch iskaza, oni se razmatraju po redosledu pojavljivanja i bira se prvi koji odgovara po tipu argumenta.

```
try {
  //...
}
catch (DeviceError &de){//...}// naveden je prvo argument tipa
                              // osnovne klase
catch (DiskError &de) {//...} // greška: ovo nikad neće biti pozvano!
```

- Obrnuti redosled predstavlja logični sled: najpre se obrađuje neki specifični izuzetak, a onda svi ostali izuzeci iste grupe:

```
try {  //... }
catch (DiskError &de)   {  // ovde obradi grešku sa diskom,
  //...
}
catch (DeviceError &de) { // a ovde sve ostale greške device error-a;
  //...
}
```

Obrada izuzetka

- Znaci **. . .** u deklaraciji argumenta catch iskaza znači da ovakav catch iskaz prihvata izuzetak bilo kog tipa.
- Ako postoji ovakav iskaz, on mora biti poslednji u listi catch iskaza jednog try bloka.
- Ako se ne nađe nijedan odgovarajući catch iskaz koji prihvata izuzetak, pretraga se nastavlja u prvom try bloku koji okružuje tekući try blok.
- Takav mehanizam predstavlja pretragu po hijerarhiji poziva, odnosno po steku poziva funkcija i blokova.
- Ako se u celom programu ne nađe ni jedan odgovarajući catch iskaz, poziva se specijalna funkcija **terminate**.
- Kada se dođe do mesta obrade izuzetka stek poziva će biti vraćen u stanje koje odgovara mestu obrade izuzetka.

Konstruktori i destruktori

- Kada se kontrola toka programa prebacuje sa mesta postavljanja izuzetka na mesto obrade izuzetka, pozivaju se destruktori svih automatskih objekata koji su konstruisani od trenutka ulaska u try blok koji odgovara mestu obrade izuzetka.
- Ako je izuzetak postavljen u toku konstruisanja nekog objekta (unutar njegovog konstruktora ili funkcija koje se pozivaju unutar konstruktora), za delimično konstruisan objekat pozvaće se destruktori samo onih članova i podobjekata osnovnih klasa koji su u potpunosti konstruisani.
- Slično, ako konstruktor nekog od elemenata niza postavi izuzetak, pozvaće se destruktori samo konstruisanih elemenata niza.

- Ovakav proces poziva destruktora automatskih objekata, koji su kreirani na putu od ulaska u try blok, do mesta postavljanja izuzetka, predstavlja opisano vraćanje steka poziva na stanje koje u potpunosti odgovara mestu obrade izuzetka.
    - Ovakav proces naziva se "odmotavanjem steka" (*stack unwinding*).

Specifikacije izuzetaka

- Deklaracija funkcije koja baca izuzetke ima opšti oblik:

```
tip ime_funkcije (lista_argumenata) throw (lista_tipova);
```

- Na primer, sledeća funkcija može direktno ili indirektno postaviti izuzetke tipa X ili Y:

```
void f () throw (X,Y) {  //...}
```

- Ako funkcija postavi izuzetak čiji tip nije naveden u throw listi u deklaraciji funkcije, pozvaće se specijalna funkcija **unexpected**.
- Sledeći kôd funkcije f:

```
void f () throw (X,Y) {  //...}
```

je ekvivalentan sa kodom:

```
void f () {   try { //...   }
  catch (X) {throw;}  catch (Y) {throw;}  catch (...) {unexpected();}
}
```

- Provera poštovanja ovih specifikacija izuzetaka ne obavlja se nikad u fazi prevođenja, već isključivo u vreme izvršavanja programa.
- Funkcija koja nema throw specifikaciju u svojoj deklaraciji može postaviti bilo koji izuzetak.
- Funkcija koja ima praznu throw specifikaciju (throw()) NE može postaviti nijedan izuzetak. Funkcija koja može postaviti izuzetak tipa X, može postaviti i izuzetak tipa koji je javno izveden iz tipa X. Specifikacija izuzetaka nije deo tipa funkcije.

Specijalne funkcije za obradu izuzetaka

- Funkcija **terminate()** poziva se u slučaju da se po postavljanju izuzetka ne nađe ni jedan catch iskaz u programu koji može da prihvati ovaj izuzetak.
- Funkcija terminate je specijalna, standardna funkcija jezika C++. Ona se poziva u sledećim slučajevima:
  1. mehanizam za obradu izuzetaka ne može da pronađe odgovarajući deo za obradu generisanog izuzetka;
  2. kada ovaj mehanizam zaključi da je stek poziva poremećen,
  3. kada se u destruktoru, koji se poziva u procesu odmotavanja steka, ponovo postavi izuzetak.
- Ova funkcija je deklarisana kao:

```
void terminate(); // poziva funkciju koja je poslednja specificirana
                  // pozivom funkcije set_terminate:
typedef void (*PFV)();
PFV set_terminate(PFV);
```

- Funkcija set_terminate postavlja novu funkciju (dostavljenu kao argument) kao funkciju koju će pozivati funkcija terminate. Funkcija set_terminate kao rezultat vraća prethodno ovako definisanu funkciju.
- Podrazumevana funkcija koja se poziva iz funkcije terminate je standardna funkcija **abort().**

Funkcija unexpected

- Funkcija **unexpected()** poziva se u slučaju da neka funkcija prekrši specifikaciju izuzetaka u svojoj deklaraciji.
- Funkcija unexpected je standardna funkcija jezika C++. Ona se poziva u slučaju da neka funkcija, direktno ili indirektno, postavi izuzetak tipa koji se ne nalazi u listi tipova throw specifikacije u njenoj deklaraciji. Ova funkcija je deklarisana kao:

```
void unexpected();
```

- Ova funkcija poziva funkciju koja je poslednja specificirana pozivom funkcije set_unexpected:

```
typedef void (*PFV)();
PFV set_unexpected(PFV);
```

- Funkcija set_unexpected postavlja novu funkciju, dostavljenu kao argument, kao funkciju koju će pozivati funkcija unexpected.
    - Funkcija set_unexpected kao rezultat vraća prethodno ovako definisanu funkciju.
- Podrazumevana funkcija koja se poziva iz funkcije unexpected je funkcija **terminate()**.
  - kako funkcija terminate podrazumevano poziva funkciju **abort()**, u navedenom slučaju greške, program se odmah prekida.

Mini primer

```
#include <iostream>
using namespace std;
class DivisionError{
  double d; public:
  DivisionError(double x) : d(x) {}
  void Handle(){cout<<endl<<"Division of "<<d<<"by zero!"<<endl;}
};

void f(double x, double y) {
  int z;
  if (y==0) throw DivisionError(x);
  else z=x/y;
  cout<<"z="<<z<<endl;
}
void main () {
  int a(10), b(0);
  try{
   f(a,b);
  }
  catch (DivisionError &de) { de.Handle(); }
  system("pause");
}
```

Lambda

- Lambda funkcija:

```
#include <algorithm>
#include <cmath>
void abssort(float* x, unsigned n) {
    std::sort(x, x + n, [](float a, float b) {
                            return (std::abs(a) < std::abs(b));
                        }
    );
}
```

- Format lambda funkcije je:
  - [nacin_korišćenja_vanjskih_promenljivih]
    = , & (po vrednosti, po referenci, respektivno)
  - ->(povratni_tip)
  - (parametri_poziva)
  - { telo funkcije }

Mini primer

```
// compile with: cl /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main(){
   vector<int> v;
   for (int i = 1; i < 10; ++i) {   v.push_back(i);   }
   int evenCount = 0;
   for_each(v.begin(), v.end(),
            [&evenCount] (int n) {
               cout << n;
               if (n % 2 == 0) {
                  cout << " is even " << endl;
                  ++evenCount;
               } else {
                  cout << " is odd " << endl;
               }
   });
   cout << "There are " << evenCount << " even numbers in the vector."
        << endl;   system("pause"); return 0;
}
```