Računske vježbe 11

Programiranje II

Projektovati klasu za obradu vektora (niza) realnih brojeva sa zadatim opsezima indeksa. Za razrješavanje konfliktnih situacija koristiti mehanizam obrade izuzetaka. Napisati glavni program za prikazivanje mogućnosti klase.

```
#include <iostream>

using namespace std;
//neispravan opseg indeksiranja
class BoundsException : public exception
{
  const char* what() const noexcept override
  {
    return "BoundsException!";
  }
};
//dodjela memorije nije uspjela
class OutOfMemoryException : public exception
{
  const char* what() const noexcept override
  {
    return "OutOfMemoryException!";
  }
};
//vektor je prazan
class EmptyException : public exception
{
  const char* what() const noexcept override
  {
    return "EmptyException";
  }
};
//indeks je izvan opsega
class IndexOutOfBoundsException : public exception
{
  const char* what() const noexcept override
  {
    return "IndexOutOfBoundsException";
  }
};
//neusaglasene duzine vektora
class InconsistentLengthException : public exception
{
  const char* what() const noexcept override
  {
    return "InconsistentLengthException";
  }
};

```

```
class Vector
{
private:
  float* arr;
  int lowerBound;
  int upperBound;
public:
  Vector() { arr = 0; lowerBound = 0; upperBound = 0; }
  Vector(int, int);
  Vector(const Vector&);
  ~Vector() { delete[]arr; arr = 0; }
  Vector& operator=(const Vector&);
  float& operator[](int) const;
  friend double operator*(const Vector&, const Vector&);
};

Vector::Vector(int lowerBound, int upperBound) : lowerBound(lowerBound), upperBound(
    upperBound)
{
  if (lowerBound > upperBound) throw BoundsException();
  if (!(arr = new float[upperBound - lowerBound + 1])) throw OutOfMemoryException();
  for (int i = 0; i < upperBound - lowerBound + 1; i++) arr[i] = 0;
}

Vector::Vector(const Vector& vec) : lowerBound(vec.lowerBound), upperBound(vec.
    upperBound)
{
  if (!(arr = new float[upperBound - lowerBound + 1])) throw OutOfMemoryException();
  for (int i = 0; i < upperBound - lowerBound + 1; i++) arr[i] = vec.arr[i];
}

Vector& Vector::operator=(const Vector& vec)
{
  if (vec.arr == 0) throw EmptyException();
  if (&vec != this)
  {
    delete[] arr;
    lowerBound = vec.lowerBound;
    upperBound = vec.upperBound;
    if (!(arr = new float[upperBound - lowerBound + 1])) throw OutOfMemoryException()
    ;
    else for (int i = 0; i < upperBound - lowerBound + 1; i++) arr[i] = vec.arr[i];
  }
  return *this;
}

float& Vector::operator[](int i) const
{
  if (!arr) throw EmptyException(); // u niz nije nista upisano, samo je izvrsen
    default konstruktor
  else if (i < lowerBound || i > upperBound) throw IndexOutOfBoundsException();
  else return arr[i - lowerBound];
}

double operator*(const Vector& v1, const Vector& v2)
{
  if (!v1.arr || !v2.arr) throw EmptyException();
  else if ((v1.upperBound - v1.lowerBound) != (v2.upperBound - v2.lowerBound)) throw
    InconsistentLengthException();
```

```
  else
  {
    double dotProduct = 0;
    for (int i = 0; i < v1.upperBound - v1.lowerBound + 1; i++)
      dotProduct += v1.arr[i] * v2.arr[i];
    return dotProduct;
  }
}

int main()
{
  while (true)
  {
    try
    {
      int upperBound, lowerBound;

      cout << "Unesite opseg indeksa prvog niza" << endl;
      cin >> lowerBound >> upperBound;
      if (cin.fail()) throw 1;
      Vector v1(lowerBound, upperBound);

      cout << "Elementi prvog niza" << endl;
      for (int i = lowerBound; i <= upperBound; i++) cin >> v1[i];

      cout << "Unesi opseg indeksa drugog niza" << endl;
      cin >> lowerBound >> upperBound;
      Vector v2(lowerBound, upperBound);

      cout << "Elementi drugog niza" << endl;
      for (int i = lowerBound; i <= upperBound; i++) cin >> v2[i];

      cout << "Skalarni proizvod dva zadata niza je " << v1 * v2 << endl;
    }
    catch (const exception& e)
    {
      cout << e.what() << endl;
    }
    catch (...) // kad ne zelimo da specificiramo ili ne znamo tip izuzetka
    {
      cout << "Kraj unosa!" << endl;
      break;
    }
  }
}
```

Konfliktne situacije odnosno greške su sastavni dio programa. Kada programski jezik nema posebnu podršku za obradu izuzetaka sva odgovornost pada na programera. Sa tim smo se susreli u programskom jeziku C, ali i u dobrom dijelu ovog kursa, sve do sada. Kako smo mi do sada obrađivali izuzetke? U funkcijama koje smo pisali obično smo imali neku dogovorenu vrijednost koja je označavala grešku ili neuspjeh izvršenja određene radnje, a koju funkcija vraća. Ovo samo po sebi nije zamorno ukoliko se radi o jednoj funkciji koju pozivamo direktno. Međutim, kao što je to čest slučaj, obično bismo imali situacije kada jedna funkcija poziva drugu itd. što dalje implicira da je svaka od ovih funkcija morala biti opterećena brigom o tome da li je negdje greška otkrivena ili ne. Ovo nije dobra programerska praksa zato što se svaka od ovih nestandardnih obrada izuzetaka mora dobro dokumentovati, istestirati itd. što ugrožava skalabilnost sistema. Jezik C++ (i gotovo svi moderni programski jezici) nam nudi efikasan mehanizam za obradu izuzetaka koji u dobroj mjeri rasterećuje programere. Kad se na nekom mjestu u programu otkrije greška

ona se prijavljuje određenim izuzetkom (engl. *exception*). Započete aktivnosti se prekidaju i upravljanje se automatski predaje rukovaocu izuzecima (engl. *exception handler*). Pojava izuzetka prijavljuje se izrazom oblika:

```
throw izuzetak
```

i govori se o **bacanju** izuzetka. Izuzeci u jeziku C++ mogu se predstaviti podacima standardnih ili klasnih tipova. Tip podatka koji se šalje rukovaocu naziva se tip izuzetka. Tip podatka koji se baca se prevashodno koristi da se odredi kom rukovaocu se ta informacija dostavlja. Drugim riječima, mi tipom izuzetka možemo reći o kojoj grešci je riječ. Takođe, ako se odlučimo da bacamo npr. cjelobrojne vrijednosti, bačenom vrijednošću možemo obavijestiti rukovaoca o kojoj grešci se radi (primjenom uslovnih naredbi) što nije dobra praksa. Nemjerljivo je konciznije ukoliko bacimo izuzetak tipa npr. *OutOfMemoryException* nego recimo *int* vrijednosti 6. Ovo 6 nam samo po sebi ništa ne znači (morali bismo pročitati u dokumentaciji) dok nam naziv ove klase nedvosmisleno govori o kakvom izuzetku je riječ. Bačeni izuzetak se hvata pomoću:

```
try
{
//naredbe koje mogu baciti izuzetak
}
catch (tip identifikator)
{
//obrada izuzetka datog tipa
}
catch (tip identifikator)
{
//obrada izuzetka datog tipa
}
...
catch (...)
{
//obrada izuzetaka svih preostalih tipova koje iznad nismo posebno obradili
}
```

gdje se sa engleskim riječima pokušaj (engl. *try*) i uhvati (engl. *catch*) jasno naglašava šta rukovalac izuzecima radi. Uočimo da **catch** ima tačno jedan parametar. Poseban slučaj jesu tri tačke **(...)** pomoću kojih označavamo da rukovalac može da uhvati izuzetke svih tipova. Ako se do napuštanja bloka naredbi nakon **try** ne dogodi izuzetak, preskaču se svi rukovaoci izuzecima. Pojavi li se neki izuzetak unutar *try* bloka, prekida se njegovo izvršavanje i odabira se rukovalac koji može da uhvati taj izuzetak, tj. onaj čiji se tip slaže se tipom nastalog izuzetka. U praksi se najčešće bacaju objekti specijalne namjene koji predstavljaju izuzetke pa smo tako u našem zadatku definisali nekoliko klasa koje predstavljaju izuzetke. Jedna od njih je:

```
class OutOfMemoryException : public exception
{
  const char* what() const noexcept override
  {
    return "OutOfMemoryException!";
  }
};
```

pa ćemo bacanjem objekta ove klase rukovaocu dati do znanja da nije bilo dovoljno memorije za alokaciju. Ova klasa nasljeđuje klasu **std::exception** koju nam nudi standardna biblioteka. Odlučili smo se za ovaj korak zato što klasa *exception* ima virtuelnu metodu *what* koja vraća string koji predstavlja opis izuzetka odnosno greške. Tumačimo sada potpis ove virtuelne metode. Ključnom riječju **override** (tehnički i nije ključna riječ, koga interesuje nek istraži) naglašavamo programeru koji čita naš kod da se radi o vrituelnoj metodi koja nadjačava virtuelnu metodu osnovne klase. Takođe, ako nam je to i bila namjera, *override* će pomoći kompajleru da nam prijavi grešku ukoliko smo slučajno pogriješili u potpisu ove metode u odnosu na

onu iz osnovne klase. Mada smo *override* preskočili da pomenemo kada smo govorili o virtuelnim metodama ovo je i dalje dobra prilika jer upravo u praksi koristimo pogodnosti nasljeđivanja. Sve funkcije se mogu podijeliti na one koje sigurno ne bacaju izuzetke i one koje možda bacaju izuzetke. Specifikator **noexcept** predstavlja način da pomognemo kompajleru da zna da napravi razliku između ove dvije vrste funkcija čime se kompajliranje optimizuje. Napomenimo da *override* i *noexcept* nisu obavezni. Pored pogodnosti koju nam nudi ova virtuelna metoda, za nasljeđivanje smo se odlučili prije svega kako bismo osigurali sljedeću vezu: naš izuzetak **je** *exception*. Kako *OutOfMemoryException* jeste *exception* onda:

```
catch (const exception& e)
{
  cout << e.what() << endl;
}
```

ovako definisanim rukovaocem osiguravamo da će svi izuzeci koji nasljeđuju *exception* biti njime uhvaćeni, uključujući i *OutOfMemoryException*. Standardna C++ praksa govori da se baca po vrijednosti, a hvata po referenci. Objekat izuzetka se uvijek kopira prije nego li stigne do rukovaoca, ali nam upravo referenca omogućava iskorišćenje pogodnosti nasljeđivanja. Dakle, ne moramo za svaki tip izuzetka konstruisati novog rukovaoca. Na predavanjima smo naučili da su **cin** i **cout** globalni objekti, a ne funkcije ili operatori. Pored očiglednih pogodnosti upisa i ispisa, objekti klase iostream i ostream (cin i cout su instance ovih klasa) posjeduju određene metode koje nam mogu biti od koristi. U našem slučaju smo koristili:

```
if (cin.fail()) throw 1;
```

metodu **fail** pomoću koje provjeravamo da li je upis bio uspješan. Kako su promjenljive u koje vršimo upis tipa *int*, unos nekog karaktera rezultovaće time da nam metoda *fail* vrati *false*. Na ovaj način zaustavljamo petlju odnosno upis. Uočite da ovu provjeru nismo vršili na ostalim mjestima što ne znači da ona tamo nije neophodna već da smo njenu upotrebu ilustrovali na samo jednom primjeru. Bacamo izuzetak tipa *int* kako bismo pokazali da izuzetak može biti standardnog tipa, a on će biti uhvaćen unutar rukovaoca kojeg smo definisali sa tri tačke zato što *int* **nije** *std::exception*. Dodatno, vrijedi objasniti kako smo provjeravali da li je za niz realnih brojeva zauzeta memorija ili ne. Naime, u podrazumijevanom konstruktoru smo vodili računa o tome da pokazivač inicijalizujemo na 0. Adresa zauzete memorije mora predstavljati neku pozitivnu vrijednost i znamo da nikada neće biti nula. Pomoću operatora **!** provjeravamo zauzetost memorije:

```
if (!arr) // ili
if (!(arr = new float[upperBound - lowerBound + 1])) // odmah prilikom alokacije
```

gdje u prvoj liniji znamo da ako je *arr* 0 logički izraz će biti tačan, a u drugoj združujemo alokaciju i provjeru jer znamo da će nam alokacija vratiti NULL pokazivač (čija je vrijednost zapravo 0) i to se svodi na slučaj iznad. Takođe, bez ovog operatora mogli smo i jesmo alokaciju provjeravati kao i u ranijim zadacima sa:

```
if (vec.arr == 0)
```

ali smo ovaj pristup izbjegavali iz prostog razloga što je duži za zapisati. Osvrnimo se još jednom na izuzetke prije kraja ove lekcije. Programeri početnici i studenti često ne razumiju svrhu postojanja mehanizma za obradu izuzetaka. To je zbog toga što im se čini da se svaka greška u programu može predvidjeti, izbjeći i valjano obraditi u kodu. Izuzeci ne postoje kako bi se od greške zaštitio loš kod! Izuzeci postoje najprije zbog situacija koje nije moguće predvidjeti, a koje utiču na rad programa odnosno čija pojava uslovljava program da promijeni tok izvršavanja. Odličan primjer za to je nestanak internet konekcije. U aplikaciji koja se oslanja na internet konekciju, gubitak te konekcije predstavlja izuzetak koji je nepredvidiv (korisnik aplikacije ulazi u lift), ali koji ne smije značiti kraj programa. Kako se ovaj izuzetak prevazilazi? Aplikacija se okreće lokalnom skladištu i čita podatke iz interne memorije koje je keširala, spremajući se za ovaj izuzetak. Lokalno skladište je prazno? Novi izuzetak, aplikacija na vrijeme nije stigla da ga napuni podacima. Kako dalje? Pa možemo putem korisničkog interfejsa korisniku dati do znanja da je ostao bez konekcije i da pokuša ponovo.