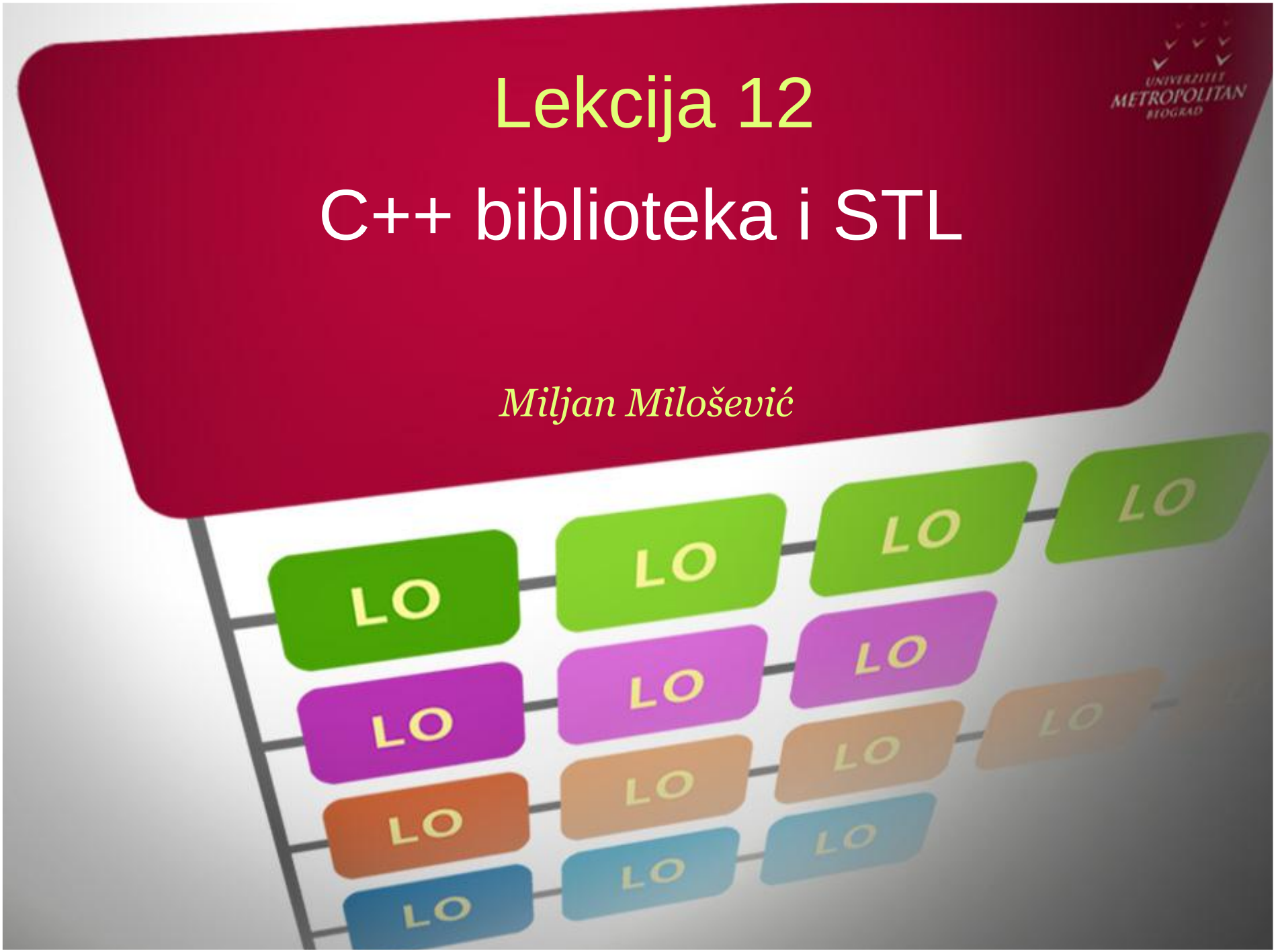
Lekcija 12

C++ biblioteka i STL

Miljan Milošević



C++ BIBLIOTEKA I STL

	01	02	03	04
Uvod	**Uvod u C++ standardnu biblioteku**	**Kontejnerske klase**	**Šabloni kontejnerskih klasa**	**Uvod u STL**
	➢ *C++ standardna biblioteka*	➢ *Uvod u kontejnerske klase* ➢ *Kreiranje kontejnerske klase za niz* ➢ *Implementacija konstruktora i destruktora* ➢ *Implementacija preklopljenih operatora i provera granice* ➢ *Promena veličine niza, ubacivanje i brisanje elemenata* ➢ *Testiranje kreirane kontejnerske klase*	➢ *Potreba za šablonima kontejnerskih klasa* ➢ *Kreiranje šablona kontejnerskih klasa*	➢ *STL – biblioteka standardnih šablona* ➢ *Uvod u STL kontejnerske klase*

C++ BIBLIOTEKA I STL

05

Sekvencionalni kontejneri

- *STL vektori*
- *Upotreba vektora*
- *STL dekovi*
- *STL liste*
- *Upotreba STL liste*

06

Asocijativni kontejneri i adapteri

- *Uvod u asocijativne kontejnere*
- *STL mapa*
- *Upotreba mapa i vektora*
- *Adaptivni kontejneri - kontejneri specijalne namene*
- *Kreiranje kontejnera za stek*

07

STL Iteratori

- *Uvod u iteratore*
- *Funkcije za rad sa iteratorima*
- *Upotreba iteratora – Kretanje kroz vektor i listu*
- *Upotreba iteratora – Kretanje kroz set*
- *Upotreba iteratora – Kretanje kroz mapu*

08

STL Algoritmi

- *Osnovi o STL algoritmima*
- *Funkcije min_element, max_element, find (i list::insert)*
- *Funkcije sort i reverse*

09

STL Stringovi

- *Klasa za stringove - std::string*
- *Osnovni pregled string klase*
- *Funkcije za rad sa string klasom*
- *Pristup karakterima stringa*
- *Pretvaranje stringa u C-string (niz karaktera)*
- *Iteratori i stringovi*

C++ BIBLIOTEKA I STL

10

Vežbe

- *Upotreba vektora – Primer1*
- *Upotreba vektora – Primer2*
- *Primer. Iteratori za ulazni i izlazni tok*
- *Primer. Vektori, iteratori i šabloni*
- *Primer. Funkcije za manipulaciju vektorima*
- *Primer. Upotreba steka*
- *Primer. Upotreba reda i reda sa prioritetom*

11

Zadaci za samostalan rad

- *Zadaci za samostalno vežbanje*

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

C ++ standardna biblioteka (C++ Standard Library) je skup klasa i funkcija koje su napisane na izvornom jeziku. C++ standardna biblioteka obezbeđuje nekoliko generičkih kontejnera, funkcija koje angažuju i manipulišu ovim kontejnerima, objekte funkcija (functors), generičke stringove i tokove (uključujući standardni ulaz/izlaz, kao i rad sa fajlovima), kao i standardne funkcije za obavljanje svakodnevnih zadataka kao što su pronalaženje kvadrata, koren broja itd...

U stvarnom životu, koristimo posude (containers) u velikom broju situacija. Vaš doručak žitarica nosite u kutiji, stranice u vašoj knjizi dolaze unutar korica knjige, a nekada imate potrebu da skladištite stare predmete u kutijama u vašem podrumu ili garaži. Slično tome, kontejnerska klasa je klasa dizajnirana za držanje i organizovanje više instanci druge klase. Postoji mnogo različitih vrsta kontejnerskih klasa, od kojih svaka ima različite prednosti, mane, i ograničenja u korišćenju.

Standardizacija C++ je donela niz promena, a jedna od najbitnijih je uvođenje STL-a, Standard Template Library. STL je jednostavno kolekcija šablona klasa i funkcija dizajniranih da omoguće bolju upotrebu kontejnerskih objekata kao što su: vektori, liste, setovi, mape, itd.

STL kontejnerske klase su komponente STL biblioteke koje se najčešće koriste u praksi. STL sadrži puno kontejnerskih klasa koje se mogu koristiti u velikom broju različitih situacija. STL obuhvata puno raznih tipova kontejnera, gde svaki tip ima svoje prednosti i nedostatke. Generalno govoreći, kontejnerske klase se grupišu u tri različite kategorije i to: sekvencionalni, asocijativni i adaptivni kontejneri.

Kao dodatak na kontejnerske klase i iteratore, STL pruža veliki broj generičkih algoritama za rad sa elementima kontejnerskih klasa. Ovi algoritmi omogućavaju stvari kao što su sortiranje, pretraga, ubacivanje, promena redosleda, brisanje i kopiranje elemenata kontejnerske klase.

Standardna biblioteka sadrži mnogo korisnih klasa ali najviše korišćena klasa je verovatno std::string koju smo već pomenuli u okviru C++ kursa, kao i veliki broj njenih funkcija. std::string je klasa koja sadrži veliki broja funkcija za dodelu, poređenje i izmenu stringova.

Uvod u C++ standardnu biblioteku

Standardna C++ biblioteka, biblioteka klasa, biblioteka funkcija

- *C++ standardna biblioteka*

01

C++ STANDARDNA BIBLIOTEKA

C ++ standardna biblioteka (C++ Standard Library) je skup klasa i funkcija koje su napisane na izvornom jeziku. Sastoji se iz standardne biblioteke fukcija i biblioteke OO klasa

C ++ standardna biblioteka (C++ Standard Library) je skup klasa i funkcija koje su napisane na izvornom jeziku. C++ standardna biblioteka obezbeđuje nekoliko generičkih kontejnera, funkcija koje angažuju i manipulišu ovim kontejnerima, objekte funkcija (functors), generičke stringove i tokove (uključujući standardni ulaz/izlaz, a i rad sa fajlovima), kao i standardne funkcije za obavljanje svakodnevnih zadataka kao što su pronalaženje kvadrata, koren broja itd... Svojstva C ++ standardne biblioteke su smeštena u imenskom prostoru std. C++ standardna biblioteka se može podeliti u dva dela:

- **Standardna biblioteka funkcija** (The Standard Function Library): Ova biblioteka se sastoji iz funkcija opšte namene koje su samostalne i koje nisu deo klasa, a nasleđena je iz C-a.
- **Biblioteka OO klasa** (The Object Oriented Class Library): Predstavlja kolekciju klasa i odgovarajućih funkcija članica.

Standarna C++ biblioteka sadrži sve elemente C biblioteke, uz male izmene koje obezbeđuju sigurnost upotrebe tipova podataka (type safety).

Standardna biblioteka funkcija se sastoji iz sledećih kategorija:

- Ulazno/izlazne funkcije I/O
- Funkcije za rad sa stringovima i karakterima
- Matematičke funkcije
- Funkcije za rad sa vremenom i datumom
- Funkcije za dinamičko upravljanje memorijom
- Ostale funkcije

Standardna C++ OO biblioteka definiše dodatni skup klasa koje pružaju podršku mnogobrojnim svakodnevnim aktivnostima, uključujući ulaz/izlaz, rad sa stringovima i procesiranje brojeve. Ova biblioteka se sastoji iz:

- Standardnih C++ klasa za ulaz/izlaz
- Klasa za rad sa stringovima (string)
- Klasa za rad sa brojevima
- STL kontejnerskih klasa
- STL algoritama
- STL funkcijskih objekata
- STL iteratora
- STL alokatora
- Klasa za upravljanje izuzecima
- Biblioteka za lokalizaciju
- Biblioteka za raznoraznu podršku

Kontejnerske klase

Kontejneri, kontejnerske klase, funkcije kontejnera, kontejner za niz

- *Uvod u kontejnerske klase*
- *Kreiranje kontejnerske klase za niz*
- *Implementacija konstruktora i destruktora*
- *Implementacija preklopljenih operatora i provera granice*
- *Promena veličine niza, ubacivanje i brisanje elemenata*
- *Testiranje kreirane kontejnerske klase*

02

UVOD U KONTEJNERSKE KLASE

Kontejnerska klasa je klasa dizajnirana za držanje i organizovanje više instanci druge klase. Obično se javlja u dva oblika i to kao vrednosni odnosno referencni kontejneri

U stvarnom životu, koristimo posude (containers) u velikom broju situacija. Vaš doručak žitarica nosite u kutiji, stranice u vašoj knjizi dolaze unutar korica knjige, a nekada imate potrebu da skladištite stare predmete u kutijama u vašem podrumu ili garaži. Bez posuda, rad nad nekim od prethodnih primera bi bio nezamisliv, nastao bi haos. Stoga je glavna prednost kontejnera u sposobnosti da organizuje i skladišti predmete koji se ubacuju u njega.

Slično tome, kontejnerska klasa je klasa dizajnirana za držanje i organizovanje više instanci druge klase. Postoji mnogo različitih vrsta kontejnerskih klasa, od kojih svaka ima različite prednosti, mane, i ograničenja u korišćenju. Do sada najčešće korišćeni kontejner u programiranju je bio niz, koji ste već videli u mnogo primera. Iako C ++ ima ugrađenu funkcionalnost za rad sa nizovima, programeri će često koristiti kontejnersku klasu za niz umesto standarnog niza zbog dodatnih pogodnosti koje pruža. Za razliku od ugrađenih nizova, kontejnerska klasa za niz podržava dinamičku promenu veličine (kada se elementi dodaju ili uklanjaju) i proveru granica (bounds-checking). Ovo ne samo da kontejnerske klase nizova čini pogodnije za korišćenje od običnih nizova, već i sigurnije.

Kontejnerska klasa se obično javlja u dva oblika. Prvi oblik su takozvani vrednosni kontejneri kreirani po principi kompozicije, i čuvaju kopije objekata koji ulaze u sastav kontejnera (a time su odgovorni za stvaranje i uništavanje te kopije). Drugi oblik su referencni kontejneri koji su kreirani po principu agregacije (skupa) i koji čuvaju pokazivače ili reference na druge objekte (i na taj način nisu odgovorni za stvaranje ili uništavanje tih objekata).

Za razliku od stvarnog života, gde kontejneri mogu držati sve što ubacite u njih, u C++-u, kontejneri obično skladište samo jednu vrstu podataka. Na primer, ako imate niz celih brojeva, on će samo držati cele brojeve. Za razliku od nekih drugih jezika, C++ generalno ne dozvoljava da se mešaju tipovi podatka u jednom kontejneru. Ako želite da imate jednu kontejnersku klasu koja sadrži cele brojeve a drugu da sadrži realne (double), moraćete da napiše dva odvojena kontejnera za ovaj poduhvat (ili jednostavno koristite šablone, koji su napredna C ++ funkcija). Uprkos ograničenjima njihovog korišćenja, kontejneri su neizmerno korisni jer čine programiranje lakšim, sigurnijim i bržim.

KREIRANJE KONTEJNERSKE KLASE ZA NIZ

Iako C ++ ima ugrađenu funkcionalnost za rad sa nizovima, programeri će često koristiti kontejnersku klasu za niz umesto standarnog niza

Kontejnerska klasa obično implementira samo minimalnu funcinalnost. Najveći broj kontejnera će uglavnom sadržati funkcije za:

- Kreiranje i pražnjenje kontejnera (pomoću konstruktora)
- Ubacivanje novog objekta u kontejner
- Brisanje objekta iz kontejnera
- Izveštavanje o broju objekta koji su trenutno u kontejneru
- Brisanje svih objekata iz kontejnera
- Pristup objektima kontejnera
- Sortiranje elemenata (opciono).

Ponekad će iz određenih kontejnerskih klasa biti izostavljena neka od ovih funkcionalnosti. Na primer, kontejnerske klase za nizove uglavnom nemaju funkcije za ubacivanje i brisanje elemenata jer su ove operacije spore i programer stoga neće ohrabriti njihovu upotrebu.

U ovom primeru ćemo napisati klasu za celobrojni niz u kojoj će biti implementiran najveći deo funkcionalnosti koju jedan kontejner treba da sadrži.

Kreiraćemo klasu IntArray koja će da bude vrednosni kontejner, tj. sadržaće kopije objekata koji su organizovani u njemu. Definiciju kontejnerske klase ćemo kreirati u IntArray.h fajlu:

```
#ifndef INTARRAY_H
#define INTARRAY_H

class IntArray
{
};
#endif
```

Naša klasa IntArray će čuvati dve vrednosti: sam niz i njegovu veličinu. S obzirom da želimo da naš niz može da se menja po veličini, koristićemo dinamičko alociranje memorije pa nam u tu svrhu treba i pokazivač:

```
#ifndef INTARRAY_H
#define INTARRAY_H

class IntArray
{
private:
          int m_nLength;
          int *m_pnData;
};
#endif
```

IMPLEMENTACIJA KONSTRUKTORA I DESTRUKTORA

Pri radu sa kontejnerskim klasama obično se koriste dinamički nizovi kao objekti kontejnera pa je stoga neophodno u destruktoru obezbediti odgovarajuće brisanje dinamički alocirane memorije

Sada ćemo dodati konstruktore koji omogućavaju kreiranje objekta klase IntArrays. Dodaćemo dva konstruktora: prvi, podrazumevajući, koji kreira prazan niz i drugi, konstruktor sa parametrom, koji će omogućiti kreiranje niza prema zadatoj veličini. Sada naša klasa ima sledeći oblik:

```
#ifndef INTARRAY_H
#define INTARRAY_H

class IntArray
{
private:
            int m_nLength;
            int *m_pnData;

public:
            IntArray()
            {
                        m_nLength = 0;
                        m_pnData = 0;
            }

            IntArray(int nLength)
            {
                        m_pnData = new int[nLength];
                        m_nLength = nLength;
            }
};
#endif
```

Takođe nam trebaju i funkcije koje će po potrebi da očiste objekat klase IntArrays. Prvo, kreiraćemo destruktor koji će jednostavno da obriše dinamički alociran prostor. Drugo, kreiraćemo funkciju pod nazivom Erase() koja će po potrebi da briše niz i postavi dužinu na nulu:

```
   ~IntArray()
    {
        delete[] m_pnData;
    }

    void Erase()
    {
        delete[] m_pnData;
        // We need to make sure we set m_pnData to 0
here, otherwise it will
        // be left pointing at deallocated memory!
        m_pnData = 0;
        m_nLength = 0;
    }
```

IMPLEMENTACIJA PREKLOPLJENIH OPERATORA I PROVERA GRANICE

Kod kontejnerskih klasa za niz se koristi preklapanje operatora indeksiranja [] kako bi mogli direktno, kao kod običnih nizova, da pristupimo ili izmenimo vrednost elementima kontejnera

Neophodno je da uradimo i preklapanje operatora indeksiranja [] kako bi mogli da pristupimo elementima niza. Takođe treba da obezbedimo proveru granica kako bi bili sigurni da koristimo validne indekse niza (sprečavanje izlaska iz opsega). U tu svrhu ćemo koristiti funkciju assert(). O assert funkcijama ćemo nešto više reći u sledećoj lekciji. Za sada ćemo reći da su objave, assert, pretprocesorske makro funkcije koje ispituju uslovni izraz. Ukoliko je uslovni izraz tačan neće se desiti ništa. Međutim, ukoliko je uslov netačan prekinuće se izvršavanje programa i objaviće se poruka o nastaloj grešci.

Da se vratimo sada na naš primer: osim funkcija koje smo pomenuli takođe ćemo dodati pristupne funkcije koje vraćaju dužinu niza. U ovom trenutku imamo funkcionalan kontejner IntArray koji možemo da koristimo. Imamo mogućnost da alociramo niz odgovarajuće dužine, i možemo da koristimo operator indeksiranja [] kako bi očitali ili izmenili vrednost elementima niza. Naša klasa sada izgleda ovako:

```
#ifndef INTARRAY_H
#define INTARRAY_H
#include <assert.h> // for assert()

class IntArray
{
private:
 int m_nLength;
 int *m_pnData;
public:
 IntArray()
 {
  m_nLength = 0;
  m_pnData = 0;
 }
 IntArray(int nLength)
 {
  m_pnData = new int[nLength];
  m_nLength = nLength;
 }
 ~IntArray()
 {
  delete[] m_pnData;
 }
```

IMPLEMENTACIJA PREKLOPLJENIH OPERATORA I PROVERA GRANICE

Kod kontejnerskih klasa za niz se koristi preklapanje operatora indeksiranja [] kako bi mogli direktno, kao kod običnih nizova, da pristupimo ili izmenimo vrednost elementima kontejnera

Neophodno je da uradimo i preklapanje operatora indeksiranja [] kako bi mogli da pristupimo elementima niza. Takođe treba da obezbedimo proveru granica kako bi bili sigurni da koristimo validne indekse niza (sprečavanje izlaska iz opsega). U tu svrhu ćemo koristiti funkciju assert(). O assert funkcijama ćemo nešto više reći u sledećoj lekciji. Za sada ćemo reći da su objave, assert, pretprocesorske makro funkcije koje ispituju uslovni izraz. Ukoliko je uslovni izraz tačan neće se desiti ništa. Međutim, ukoliko je uslov netačan prekinuće se izvršavanje programa i objaviće se poruka o nastaloj grešci.

Da se vratimo sada na naš primer: osim funkcija koje smo pomenuli takođe ćemo dodati pristupne funkcije koje vraćaju dužinu niza. U ovom trenutku imamo funkcionalan kontejner IntArray koji možemo da koristimo. Imamo mogućnost da alociramo niz odgovarajuće dužine, i možemo da koristimo operator indeksiranja [] kako bi očitali ili izmenili vrednost elementima niza. Naša klasa sada izgleda ovako:

```
 void Erase()
 {
  delete[] m_pnData;
  m_pnData = 0;
  m_nLength = 0;
 }
 int& operator[](int nIndex)
 {
  assert(nIndex >= 0 && nIndex < m_nLength);
  return m_pnData[nIndex];
 }
 int GetLength() { return m_nLength; }
};
```

PROMENA VELIČINE NIZA, UBACIVANJE I BRISANJE ELEMENATA

Iako je to navedeno u ovom primeru, uobičajene kontejnerske klase za nizove uglavnom nemaju funkcije za ubacivanje i brisanje elemenata u sredinu niza jer su ove operacije dosta spore

Postoji, međutim, još određen broj stvari koje nedostaju našoj klasi. Mi i dalje ne možemo da promenimo veličinu niza, ne možemo da ubacimo ili obrišemo elemente iz njega, odnosno da ga sortiramo. Stoga ćemo dodati deo koda koji nam omogućava da izmenimo veličinu niza. Napisaćemo dve različite funkcije. Prva funkcija Reallocate() će obrisati sve postojeće elemente niza i zatim će izvršiti promenu veličine niza. Ova funkcija radi dosta brzo. Druga funkcija Resize() će zadržati postojeće elemente i izvršiće proširenje niza ali će biti malo sporija.

```
void Reallocate(int nNewLength)
{
    Erase();
     if (nNewLength <= 0)
        return;
    m_pnData = new int[nNewLength];
    m_nLength = nNewLength;
}
```

```
    void Resize(int nNewLength)
    {
        if (nNewLength <= 0)
        {
            Erase();
            return;
        }
        int *pnData = new int[nNewLength];
        if (m_nLength > 0)
        {
            int nElementsToCopy = (nNewLength >
m_nLength) ? m_nLength : nNewLength;
            for (int nIndex=0; nIndex <
nElementsToCopy; nIndex++)
                pnData[nIndex] =
m_pnData[nIndex];
        }
        delete[] m_pnData;
        m_pnData = pnData;
        m_nLength = nNewLength;
    }
```

PROMENA VELIČINE NIZA, UBACIVANJE I BRISANJE ELEMENATA

Iako je to navedeno u ovom primeru, uobičajene kontejnerske klase za nizove uglavnom nemaju funkcije za ubacivanje i brisanje elemenata u sredinu niza jer su ove operacije dosta spore

Mnoge implementacije kontejnerskih klasa će se ovde završiti. Međutim, za svaki slučaj, u nastavku je dat i deo koda koji vrši umetanje i brisanje elemenata niza. Obe implementacije su slične funkciji Resize().

```
void InsertBefore(int nValue, int nIndex)
    {
        assert(nIndex >= 0 && nIndex <= m_nLength);

        int *pnData = new int[m_nLength+1];

        // Copy all of the elements up to the index
        for (int nBefore=0; nBefore < nIndex;
nBefore++)
            pnData[nBefore] = m_pnData[nBefore];

        // Insert our new element into the new array
        pnData[nIndex] = nValue;

        // Copy all of the values after the inserted
element
        for (int nAfter=nIndex; nAfter < m_nLength;
nAfter++)
            pnData[nAfter+1] = m_pnData[nAfter];

        // Finally, delete the old array, and use
the new array instead
        delete[] m_pnData;
        m_pnData = pnData;
        m_nLength += 1;
    }
```

```
    void Remove(int nIndex)
    {
        // Sanity check our nIndex value
        assert(nIndex >= 0 && nIndex < m_nLength);

        // First create a new array one element
smaller than the old array
        int *pnData = new int[m_nLength-1];

        // Copy all of the elements up to the index
        for (int nBefore=0; nBefore < nIndex;
nBefore++)
            pnData[nBefore] = m_pnData[nBefore];

        // Copy all of the values after the inserted
element
        for (int nAfter=nIndex+1; nAfter <
m_nLength; nAfter++)
            pnData[nAfter-1] = m_pnData[nAfter];

        // Finally, delete the old array, and use
the new array instead
        delete[] m_pnData;
        m_pnData = pnData;
        m_nLength -= 1;
    }

    // A couple of additional functions just for
convenience
    void InsertAtBeginning(int nValue) {
InsertBefore(nValue, 0); }
    void InsertAtEnd(int nValue) {
InsertBefore(nValue, m_nLength); }
```

TESTIRANJE KREIRANE KONTEJNERSKE KLASE

Jednom ispravno napisana kontejnerska klasa može biti korišćena koliko god puta je to neophodno bez potrebe za dodatnim izmenama u samoj klasi

Možemo sada da napišemo i glavni program kako bi proverili da li naša kontejnerska klasa radi ispravno:

```
#include <iostream>
#include "IntArray.h"

using namespace std;

int main()
{
 // Declare an array with 10 elements
 IntArray cArray(10);

 // Fill the array with numbers 1 through 10
 for (int i=0; i<10; i++)
  cArray[i] = i+1;

 // Resize the array to 8 elements
 cArray.Resize(8);

 // Insert the number 20 before the 5th element
 cArray.InsertBefore(20, 5);
```

Rezultat programa biće:

```
40 1 2 3 5 20 6 7 8 30
```

Iako pisanje kontejnerskih klasa izgleda kao prilično kompleksan posao, dobra stvar je ta da kontejnerske klase pišete samo jednom. Stoga, jednom kad je kontejnerska klasa napisana da bude funkcionalna, nju je moguće koristiti koliko god je puta to neophodno bez potrebe za dodatnim izmenama u njoj.

TESTIRANJE KREIRANE KONTEJNERSKE KLASE

Jednom ispravno napisana kontejnerska klasa može biti korišćena koliko god puta je to neophodno bez potrebe za dodatnim izmenama u samoj klasi

Možemo sada da napišemo i glavni program kako bi proverili da li naša kontejnerska klasa radi ispravno:

```
 // Remove the 3rd element
 cArray.Remove(3);

 // Add 30 and 40 to the end and beginning
 cArray.InsertAtEnd(30);
 cArray.InsertAtBeginning(40);

 // Print out all the numbers
 for (int j=0; j<cArray.GetLength(); j++)
  cout << cArray[j] << " ";

 return 0;
}
```

Rezultat programa biće:

```
40 1 2 3 5 20 6 7 8 30
```

Iako pisanje kontejnerskih klasa izgleda kao prilično kompleksan posao, dobra stvar je ta da kontejnerske klase pišete samo jednom. Stoga, jednom kad je kontejnerska klasa napisana da bude funkcionalna, nju je moguće koristiti koliko god je puta to neophodno bez potrebe za dodatnim izmenama u njoj.

Šabloni kontejnerskih klasa

Šabloni, kontejnerske klase, šabloni kontejnera

- *Potreba za šablonima kontejnerskih klasa*
- *Kreiranje šablona kontejnerskih klasa*

03

POTREBA ZA ŠABLONIMA KONTEJNERSKIH KLASA

Iako sintaksa šablona ponekada deluje čudno, šabloni klasa su jedan od najboljih i najkorišćenijih delova C++ jezika

Šabloni klasa su idealni za kreiranje kontejnerskih klasa, jer je veoma poželjno da jedna kontejnerska klasa može da radi sa različitim tipovima podataka (naravno, ne može istovremeno da skladišti različite tipove). Šabloni vam ovo omogućavaju bez ponovnog pisanja koda za svaki tip pojedinačno. Iako sintaksa deluje ružno, šabloni klasa su jedan od najboljih i najkorišćenijih delova C++ jezika.

U prethodnoj sekciji smo naučili kako da koristimo kompoziciju u cilju implementacije klasa koje sadrže višestruke instance drugih klasa. Uzećemo sada uprošćenu verziju prethodno generisane klase IntArray da bi opisali koncept korišćenja šablona kod kontejnerskih klasa.

Dok ova klasa obezbeđuje jednostavan način za kreiranje niza celih brojeva, nekada je potrebno da imamo i niz realnih brojeva ili karaktera. Zamislite kako bi to mogli da uradimo? Ukoliko koristimo tradicionalni metod razmišljanja u programiranju, verovatno bi smo kreirali potpuno novu klasu! U nastavku je dat primer kontejnerske klase DoubleArray koja će da služi kao niz realnih brojeva.

```
#ifndef DOUBLEARRAY_H
#define DOUBLEARRAY_H

#include <assert.h> // for assert()

class DoubleArray
{
private:
    int m_nLength;
    double *m_pdData;
public:
    DoubleArray()
    {
        m_nLength = 0;
        m_pdData= 0;
    }
    DoubleArray(int nLength)
    {
        m_pdData= new double[nLength];
        m_nLength = nLength;
    }
    ~DoubleArray()
    {
        delete[] m_pdData;
    }
```

POTREBA ZA ŠABLONIMA KONTEJNERSKIH KLASA

Iako sintaksa šablona ponekada deluje čudno, šabloni klasa su jedan od najboljih i najkorišćenijih delova C++ jezika

Šabloni klasa su idealni za kreiranje kontejnerskih klasa, jer je veoma poželjno da jedna kontejnerska klasa može da radi sa različitim tipovima podataka (naravno, ne može istovremeno da skladišti različite tipove). Šabloni vam ovo omogućavaju bez ponovnog pisanja koda za svaki tip pojedinačno. Iako sintaksa deluje ružno, šabloni klasa su jedan od najboljih i najkorišćenijih delova C++ jezika.

U prethodnoj sekciji smo naučili kako da koristimo kompoziciju u cilju implementacije klasa koje sadrže višestruke instance drugih klasa. Uzećemo sada uprošćenu verziju prethodno generisane klase IntArray da bi opisali koncept korišćenja šablona kod kontejnerskih klasa.

Dok ova klasa obezbeđuje jednostavan način za kreiranje niza celih brojeva, nekada je potrebno da imamo i niz realnih brojeva ili karaktera. Zamislite kako bi to mogli da uradimo? Ukoliko koristimo tradicionalni metod razmišljanja u programiranju, verovatno bi smo kreirali potpuno novu klasu! U nastavku je dat primer kontejnerske klase DoubleArray koja će da služi kao niz realnih brojeva.

```
    void Erase()
    {
        delete[] m_pdData;
        m_pdData= 0;
        m_nLength = 0;
    }
    double& operator[](int nIndex)
    {
        assert(nIndex >= 0 && nIndex <
m_nLength);
        return m_pdData[nIndex];
    }
    int GetLength() { return m_nLength; }
};
```

KREIRANJE ŠABLONA KONTEJNERSKIH KLASA

Šabloni klasa su idealni za kreiranje kontejnerskih klasa, jer je veoma poželjno da jedna kontejnerska klasa može da radi sa različitim tipovima podataka

Možemo primetititi da je izvorni kod klasa IntArray i DoubleArray gotovo identičan. Kao što pretpostavljate, ovo je oblast gde šabloni dolaze do izražaja kako bi kreirali opštu šablonsku klasu umesto da koristimo posebne klase koje su ograničene samo na jedan tip podatka. U nastavku je dat kod šablona klase Array:

```
template <typename T>
class Array
{
private:
    int m_nLength;
    T *m_ptData;
public:
    Array()
    {
        m_nLength = 0;
        m_ptData = 0;
    }
    Array(int nLength)
    {
        m_ptData= new T[nLength];
        m_nLength = nLength;
    }
    ~Array()
    {
        delete[] m_ptData;
    }
    ~Array()
    {
        delete[] m_ptData;
    }
    void Erase()
    {
        delete[] m_ptData;
        m_ptData= 0;
        m_nLength = 0;
    }
    T& operator[](int nIndex)
    {
        assert(nIndex >= 0 && nIndex <
m_nLength);
        return m_ptData[nIndex];
    }
    int GetLength();
};

template <typename T>
```

KREIRANJE ŠABLONA KONTEJNERSKIH KLASA

Šabloni klasa su idealni za kreiranje kontejnerskih klasa, jer je veoma poželjno da jedna kontejnerska klasa može da radi sa različitim tipovima podataka

Kao što se može primetiti, ova verzija je gotovo identična klasi IntArray, osim što smo dodali deklaraciju šablona (template) i promenili tip podatka elemenata niza iz int u T. Primetimo da smo funkciju GetLength() definisali izvan tela klase. Kao što smo pomenuli u prethodnoj lekciji, svaki šablon funkcije deklarisan van tela klase mora da sadrži šablonsku deklaraciju template <typename T>. Takođe, treba primetiti da je ime šablonske klase Array<T> a ne Array. Array može jedino da se odnosi na ne-šablonsku verziju klase koja se zove Array. U nastavku je kratak primer koji demonstrira upotrebu šablona kontejnerske klase Array:

```
int main()
{
 Array<int> anArray(12);
 Array<double> adArray(12);

 for (int nCount = 0; nCount < 12; nCount++)
 {
  anArray[nCount] = nCount;
  adArray[nCount] = nCount + 0.5;
 }

 for (int nCount = 11; nCount >= 0; nCount--)
 {
  std::cout << anArray[nCount] << "\t"
   << adArray[nCount] << std::endl;
 }
 return 0;
```

Uvod u STL

STL, biblioteka standardnih šablona, kolekcije klasa, kontejneri

- *STL – biblioteka standardnih šablona*
- *Uvod u STL kontejnerske klase*

04

STL – BIBLIOTEKA STANDARDNIH ŠABLONA

STL je jednostavno kolekcija šablona klasa i funkcija dizajniranih da omoguće bolju upotrebu kontejnerskih objekata kao što su: vektori, liste, setovi, mape, itd...

Standardizacija C++-a je donela niz promena, a jedna od najbitnijih je uvođenje STL-a, Standard Template Library. STL je jednostavno kolekcija šablona klasa i funkcija dizajniranih da omoguće bolju upotrebu kontejnerskih objekata kao što su: vektori, liste, setovi, mape, itd. Klase koje mogu biti definisane iz ovih šablona se naravno nazivaju kontejnerske klase. Takođe smo spomenuli da su šabloni klasa mehanizmi koji generišu kontejnerske klase. U nastavku je dat prikaz osnovnih komponenti STL-a:

Komponenta	Opis
Kontejneri	Kontejneri se koriste za upravljanje kolekcijom podataka određenog tipa. Postoji nekoliko različitih vrsta kontejnera kao što su: redovi, dekovi, liste, vektori, mape, itd...
Algoritmi	Algoritmi se primenjuju na kontejnerima. Oni omogućavaju inicijalizaciu, sortiranje, pretraživanje i transformaciju kontejenera.
Iteratori	Iteratori se koriste za putovanje kroz elemente kolekcije objekata. Ove kolekcije mogu biti kontejneri ili podskupovi kontejnera.

Slika-1 Osnovni delovi standardne C++ biblioteke šablona (STL-a)

Svaka od prethodnih komponenti STL-a sadrži bogat skup predefinisanih funkcija koje nam pomažu da veoma komplikovane probleme rešavamo na jednostavan način.

UVOD U STL KONTEJNERSKE KLASE

STL kontejnerske klase su komponente STL biblioteke koje se najčešće koriste u praksi. Postoje tri kategorije kontejnerskih klasa i to: sekvencionalni, asocijativni i adaptivni kontejneri

STL kontejnerske klase su komponente STL biblioteke koje se najčešće koriste u praksi. STL sadrži puno kontejnerskih klasa koje se mogu koristiti u velikom broju različitih situacija. Već smo spomenuli da je kontejner struktura podataka koja se sastoji od drugih objekata (po principu kompozicije ili agregacije). Ovi objekti u okviru kontejnera se zovu elementi kontejnera, a treba napomenuti da svi elementi nekog kontejnera moraju biti istog tipa.

Već smo videli kako je moguće generisati kontejnersku klasu za niz. Ovakva klasa je naravno standardni deo STL-a, i omogućuje brz pristup i brzo memorisanje svojih elemenata, gde elementi mogu biti objekti. Nedostatak kontejnera niza je taj što ima fiksnu i unapred specificiranu dužinu (broj elemenata u nizu), a ubacivanje elemenata u niz može biti komplikovano i sporo. Međutim, STL obuhvata puno raznih tipova kontejnera, gde svaki tip ima svoje prednosti i nedostatke. Generalno govoreći, kontejnerske klase se grupišu u tri različite kategorije i to: sekvencionalni, asocijativni i adaptivni kontejneri. U nastavku će biti opisana svaka od ovih kategorija.

Sekvencionalni kontejneri

Sekvence su kontejnerske klase koje održavaju redosled elemenata u kontejneru. Osnovna karakteristika sekvencionalnih kontejnera je da možete da izaberete poziciju gde želite da ubacite element. Najčešći primer sekvencionalnih kontejnera je niz: ako ubacite četiri elementa u matricu, elementi će zadržati redosled kako ste ih ubacili. STL ima 3 sekvencionalna kontejnera i to: vektor, dek i listu.

Sekvencionalni kontejneri

STL, kontejneri, sekvence, vektori, dekovi, liste

- *STL vektori*
- *Upotreba vektora*
- *STL dekovi*
- *STL liste*
- *Upotreba STL liste*

05

STL VEKTORI

Vektori u STL biblioteci se odnose na dinamičke nizove koji su sposobni da po potrebi rastu koliko je potrebno da bi se u njih ubacili dodatni elementi

Vektor (*vector*) je alternativa nizu, ali prednost mu je da njegova dimenzija tj. dužina može da se menja po potrebi u programu. Dakle, u suštini, niz je matematički gledano vektor koji ima fiksnu dužinu. Vektori mogu da se kreiraju za bilo koji tip podataka uključujući i objekte. Elementi vektora su, kao i elementi niza, numerisani počev od nule.

Ako posmatrate sa stanovišta fizike, verovatno razmišljate o vektorima kao entitetima koji imaju pravac i dužinu. Međutim, u STL biblioteci vektori se odnose na dinamičke nizove koji su sposobni da po potrebi rastu koliko je potrebno da bi se u njih ubacili dodatni elementi. Klasa vector omogućava slučajni pristup svojim elementima preko operatora [], a treba napomenuti da je ubacivanje i vađenje elemenata sa kraja vektora veoma brza operacija. Da bi se koristili vektori u nekom programu potrebno je ubaciti instrukciju

```
#include <vector>
```

koja uključuje <vector> biblioteku iz C++ biblioteke (C++ Library). Ova biblioteka sadrži niz predefinisanih metoda. Deklaracija za kreiranje vektora ima sledeću sintaksu:

```
vector <data-type> vector-name (size);
```

pri čemu se formira vektor od size elemenata koji će inicijalno imati sve nule. Opciono, vektor se može inicijalizovati i korišćenjem sledeće sintakse,

```
vector <data-type> vector-name (size, value);
```

gde je *value* inicijalna vrednost elemenata vektora. Npr. neki celobrojni vektor ima automatski inicijalizovane vrednosti elemenata jednake vrednosti 0. Ako želimo, može se umesto 0 zadati druga inicijalna vrednost. Metode koje rade sa vektorima se jednostavno pozivaju tako što se dodaju imenu vektora za kojim sledi tačka-operator:

```
vector_name.method()
```

Postoji veliki broj funkcija (metoda) za rad sa vektorima. Neke od metoda su:

- at(elemInd) - uzima vrednost elementa
- back() - uzima vrednost krajnjeg elementa
- clear() - briše vektor
- empty() - vraća 1 ako je vektor prazan, a vraća 0 ako nije prazan
- front() - uzima vrednost prvog elementa
- pop_back() - sklanja krajnji element
- push_back(value) - dodaje element na kraju vektora
- size() - vraća broj elemenata.

UPOTREBA VEKTORA

Klasa vector omogućava slučajan pristup svojim elementima preko operatora [], a treba napomenuti da je ubacivanje i vađenje elemenata sa kraja vektora veoma brza operacija

U nastavku je dat primera sa vektorima, gde se koriste prethodno pobrojane metode iz klase <vector>: at(elem-number), back(), clear(), empty(), front(), pop_back(), push_back(value), size():

```
#include <iostream>
#include <vector>
using namespace std;
int main()   {
 vector <int> vect1(3,100);
 cout << vect1.size() << endl;
 vect1.push_back(7);
 vect1.push_back(8);
 vect1.push_back(9);
 cout << vect1.size() << endl;
 cout << vect1.front()<< endl;
 cout << vect1.at(1)  << endl;
 cout << vect1.back() << endl;
 vect1.pop_back();
 cout << vect1.back() << endl;
 cout << vect1.size() << endl;
 cout << vect1.empty();
 vect1.clear();
 cout << vect1.size() << endl;
 cout << vect1.empty()<< endl;
 return 0;
}
```

Izlaz programa biće:

```
3
6
100
```

Prethodni primer koristi celobrojni vektor, ali na sličan način može se formirati vektor sastavljen od objekata, jer STL omogućuje rad sa vektorom čiji su elementi objekti (npr. string objekti). U narednom programu se formira vektor u koji se zatim ubacuje 6 elemenata, a koristi se preklopljeni operator indeksiranja [] kako bi se pristupilo elementima vektora koji se zatim štampaju.

```
#include <vector>
#include <iostream>
int main()
{
            using namespace std;
            vector<int> vect;
            for (int nCount=0; nCount < 6; nCount++)
                        vect.push_back(10 - nCount);
// insert at end of array
            for (int nIndex=0; nIndex < vect.size();
nIndex++)
                        cout << vect[nIndex] << " ";
            cout << endl;
}
```

Program će proizvesti sledeći rezultat:

```
10 9 8 7 6 5
```

STL DEKOVI

Klasa deque (dek) je ustvari dvostrani red tj. klasa dvostrukog reda, implementirana kao dinamički niz koji može da raste na oba svoja kraja

Klasa deque (dek) je ustvari dvostrani red tj. klasa dvostrukog reda, implementirana kao dinamički niz koji može da raste na oba svoja kraja. U nastavku je dat primer korišćenja klase deque:

```
#include <iostream>
#include <deque>
int main()
{
	using namespace std;

	deque<int> deq;
	for (int nCount=0; nCount < 3; nCount++)
	{
		deq.push_back(nCount); // insert at end of array
		deq.push_front(10 - nCount); // insert at front of array
	}

	for (int nIndex=0; nIndex < deq.size(); nIndex++)
		cout << deq[nIndex] << " ";

	cout << endl;
}
```

Rezultat programa biće:

```
8 9 10 0 1 2
```

STL LISTE

Lista je u C++-u implementirana korišćenjem dvostruko povezane liste, gde svaki element liste sadrži pokazivač na prethodni i naredni element

STL lista (kontejner liste, list container) povezuje objekte kao Lego kockice. Naime, objekti liste mogu biti razdvojeni i zatim ponovo spojeni u bilo kom redosledu. Ovo čini listu idealnu za insertovanje objekata, sortiranje, spajanje objekata, itd. Lista (*list)* je kontejner veoma sličan nizu ili vektoru. Međutim, za razliku od vektora, lista nema eksplicitni indeks u svojoj deklaraciji, pa se nekom elementu liste ne može pristupiti korišćenjem indeksa. Lista sadrži veliki skup funkcija, kao što su npr. insert, swap, i erase. Da bi se dodao element listi koriste se funkcije push_front ili push_back, gde prva funkcija dodaje element na početak liste, a druga funkcija dodaje element na kraj liste. Ovaj kontejner takođe omogućuje programeru da automatski putuje kroz listu obavljajući neku funkciju na svakom objektu. Putovanje kroz listu se obavlja korišćenjem iteratora, gde je iterator liste u stvari pokazivač koji pokazuje na neki element u listi.

Lista je u C++-u implementirana korišćenjem dvostruko povezane liste, gde svaki element liste sadrži pokazivač na prethodni i naredni element. Lista obezbeđuje pristup samo početnom i krajnjem elementu – tako da nije moguć nasumičan pristup proizvoljnom elementu liste. Ukoliko želimo da pristupimo elementu koji se nalazi u sredini liste neophodno je da krenemo sa kraja ili sa početka i da onda doputujemo do odgovarajućeg elementa. Prednost liste je u tome što ubacivanje elementa ili brisanje predstavlja dosta brzu operaciju ako znamo koji je taj član gde treba da izvršimo ubacivanje/brisanje.

Osnovni oblik deklaracije STL liste je:

```
list<tipPodatka> nazivListe;
```

UPOTREBA STL LISTE

Lista nema eksplicitni indeks u svojoj deklaraciji, pa se korišćenjem indeksa ne može pristupiti nekom elementu liste

Sledeći STLList program koristi listu kako bi skladištio niz imena tipa string a zatim se i sortirali elementi skupa:

```
#include <list>
#include <string>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

list<string> namesLIST;

int main(int argc, char* pArgs[])
{
 // Unos stringova
 cout << "UKUCAJ IME (UKUCAJ x NA KRAJU)" <<
endl;
 while(true)
 {
  string nameSTRING;
  cin >> nameSTRING;
  if ((nameSTRING.compare("x") == 0) ||
   (nameSTRING.compare("X") == 0))
  {
   break;
  }
  namesLIST.push_back(nameSTRING);
 }
```

Ovaj program kreira promenljivu *namesLIST* kao listu string objekata. Program prvo učitava imena preko tastature, i svako ime se dodaje na kraj liste pomoću push_back() metode. Program izlazi iz petlje kad korisnik unese ime “x”. Zatim se lista imena sortira pomoću metode sort(). Program prikazuje listu sortiranih imena tako što sklanja objekte sa liste dok je ne isprazni. Izlazni rezultat iz programa je:

```
UKUCAJ IME (UKUCAJ null NA KRAJU)
ACA DEJAN ValentinA SIMA x
Sorted output: ACA DEJAN SIMA ValentinA
Press any key to continue . . .
```

Treba za kraj napomenuti da, iako se često STL stringovi ne svrstavaju u kategoriju sekvencionalnih kontejnera, oni to ustvari jesu jer je STL string implementiran kao vektor, gde su elementi vektora karakteri (tip char).

UPOTREBA STL LISTE

Lista nema eksplicitni indeks u svojoj deklaraciji, pa se korišćenjem indeksa ne može pristupiti nekom elementu liste

Sledeći STLList program koristi listu kako bi skladištio niz imena tipa string a zatim se i sortirali elementi skupa:

```
// Sortiranje
namesLIST.sort();
cout << "\nSorted output:" << endl;
while(!namesLIST.empty())
{// uzima prvi string sa liste
 string nameSTRING = namesLIST.front();
 cout << nameSTRING << endl;
 // sklanja string sa liste
 namesLIST.pop_front();
}
return 0;
}
```

Ovaj program kreira promenljivu *namesLIST* kao listu string objekata. Program prvo učitava imena preko tastature, i svako ime se dodaje na kraj liste pomoću push_back() metode. Program izlazi iz petlje kad korisnik unese ime “x”. Zatim se lista imena sortira pomoću metode sort(). Program prikazuje listu sortiranih imena tako što sklanja objekte sa liste dok je ne isprazni. Izlazni rezultat iz programa je:

```
UKUCAJ IME (UKUCAJ null NA KRAJU)
ACA DEJAN ValentinA SIMA x
Sorted output: ACA DEJAN SIMA ValentinA
Press any key to continue . . .
```

Treba za kraj napomenuti da, iako se često STL stringovi ne svrstavaju u kategoriju sekvencionalnih kontejnera, oni to ustvari jesu jer je STL string implementiran kao vektor, gde su elementi vektora karakteri (tip char).

Asocijativni kontejneri i adapteri

STL, kontejneri, asocijativni kontejneri, adapteri, ključevi

- *Uvod u asocijativne kontejnere*
- *STL mapa*
- *Upotreba mapa i vektora*
- *Adaptivni kontejneri - kontejneri specijalne namene*
- *Kreiranje kontejnera za stek*

06

UVOD U ASOCIJATIVNE KONTEJNERE

Asocijativni kontejneri obezbeđuju direktan pristup svojim elementima preko ključa (search key). Svaki asocijativni kontejner čuva svoje ključeve u sortiranom poretku

Asocijativne kontejnere možemo posmatrati kao one kontejnere koji automatski sortiraju svoje članove, kako se novi član ubaci u skup. Podrazumeva se da asocijativni kontejneri upoređuju svoje elemente korišćenjem operatora <. U asocijativne kontejnere spadaju: skupovi(set), multi-skupovi, mape i multimape, pri čemu:

- **Set -** setovi ili skupovi su kontejneri koji čuvaju jedinstvene elemente, gde duplikati nisu dozvoljeni. Elementi u skupu se automatski sortiraju prema vrednosti.
- **Multiset** - multi setovi su skupovi u kojima je dozvoljeno postojanje duplikata.
- **Map –** mape (drugačije se nazivaju asocijativni nizovi) su skupovi gde je svaki element par, pri čemu se par sastoji iz ključa i vrednosti. Ključ je podatak prema kome se vrši sortiranje i indeksiranje pridruženih vrednosti, i on mora biti jedinstven. Vrednost predstavlja stvarni podatak koji je pridružen ključu.
- **Multimap** - ili rečnici su mape koje dozvoljavaju postojanje duplikata ključeva. Rečnici u stvarnom svetu su ustvari multi-mape: ključ je reč koja može da ima više značenja. Svi ključevi su sortirani u rastućem poretku, i svaka vrednost se pretražuje preko ključa. Neka reč može da ima više značenja, pa je stoga rečnik predstavljen kao multi-mapa a ne kao obična mapa.

STL MAPA

Mape su skupovi gde je svaki element par, pri čemu se par sastoji iz ključa i vrednosti. Ključ je podatak prema kome se vrši sortiranje i indeksiranja vrednosti, i mora biti jedinstven

Mapa je tip podataka veoma sličan vektorima, s tim što se kod vektora koriste celobrojni indeksi kao npr. vectorXX[0]. Kod mape se umesto broja kao indeks može koristiti bilo koji tip podataka pa čak i klase. U nastavku je dat primer koji bliže opisuje upotrebu kontejnera tipa mapa:

```
#include <iostream>
#include <stdlib.h>
#include <map>
#include <string>
using namespace std;
int main(int argc, char *argv[])
{
          map <string, string> marriagesMAP;
          marriagesMAP["Toma"] = "Suzana";
          marriagesMAP["Han"] = "Hana";
          cout << marriagesMAP["Toma"] << endl;
          cout << marriagesMAP["Han"] << endl;
          return 0;
}
```

Ovde vidimo da se prvo deklariše mapa

```
map<string, string> marriagesMAP;
```

gde se prvo definiše tip ključeva (keys) kao tip string, a onda tip vrednosti (values) isto kao tip string. Onda se vrši memorisanje tako što se ključ stavi u zagrade a zadaje se vrednost pomoću operatora =:

```
marriages[“Toma”] = “Suzana”;
```

Učitavanje podataka se obavlja takođe preko ključa:

```
cout << marriages[“Toma”]<<endl;
```

U cilju kretanja po kontejneru, koristi se *iterator*. Iterator omogućuje da koračate korak po korak po kontejneru. Svaka kontejnerska klasa ima ugrađen tip podataka koji se zove *iterator* (o iteratoru više reči u nastavku lekcije). Npr. ako imamo mapu:

```
map<string, int>
```

onda se iterator npr. kreira na sledeći način

```
map<string, int>::iterator loopy
```

gde je *loopy* ime iteratora koji ste izabrali. Iterator je u stvari pokazivač koji pokazuje na neki element u kontejneru. Dalje, potrebno je inicijalizovati iterator da pokazuje na prvi član u kontejneru. Ovo se može uraditi pomoću funkcije *begin*. Zatim, može se pomeriti iterator na sledeći član kontejnera ako uvećamo iterator, npr.

```
loopy++;
```

Takođe, pomoću funkcije *end*, može se proveriti da li je pokazivač stigao do kraja kontejnera.

UPOTREBA MAPA I VEKTORA

Kod mape se umesto broja kao indeks može koristiti bilo koji tip podataka pa čak i klasa. Kretanje kroz mapu se vrši korišćenjem posebnog elementa STL biblioteke koji se naziva iterator

Pogledajmo sada sledeći primer upotrebe mapa i vektora:

```
#include <iostream>
#include <stdlib.h>
#include <map>
#include <vector>
#include <string>
using namespace std;

int main(int argc, char *argv[])
{
 // Iterating through a map
 map<string, int> WordsMAP;
 WordsMAP["ten"] = 10;
 WordsMAP["twenty"] = 20;
 WordsMAP["thirty"] = 30;
 map<string, int>::iterator loopy =
WordsMAP.begin();

 while (loopy != WordsMAP.end())
 {
  cout << loopy->first << " ";
  cout << loopy->second << endl;
  loopy++;
 }
```

Kao što vidimo, u ovom primeru se prvo kreira mapa pod nazivom WordsMAP čija je inicijalna dužina nula. Pri dodavanju novih elemenata u mapu, automatski se veličina mape proširuje kako bi mogla da prihvati nove elemente. Kretanje kroz mapu, kao što smo spomenuli, se vrši korišćenjem iteratora. U ovom, slučaju, koristimo while petlju koja obezbeđuje kretanje iteratora od prvog do poslednjeg elementa u mapi, i odgovarajuću štampu na ekranu. U drugom delu programa se kreira vektor koji će da služi kao niz stringova. U ovaj vektor zatim ubacujemo 5 elemenata što automatski utiče na prišireњe njegove veličine. Kretanje kroz vektor se takođe vrši korišćenjem iteratora, pomoću koga pristupamo odgovarajućem elementu a zatim ga i štampamo na ekran. Iteratori će naravno biti detaljno opisani u nastavku lekcije.

UPOTREBA MAPA I VEKTORA

Kod mape se umesto broja kao indeks može koristiti bilo koji tip podataka pa čak i klasa. Kretanje kroz mapu se vrši korišćenjem posebnog elementa STL biblioteke koji se naziva iterator

Pogledajmo sada sledeći primer upotrebe mapa i vektora:

```
 // Iterating through a vector
 vector<string> WordsVEC;
 WordsVEC.push_back("hello");
 WordsVEC.push_back("HI");
 WordsVEC.push_back("ladies");
 WordsVEC.push_back("PLUS");
 WordsVEC.push_back("OTHERS");
 vector<string>::iterator vectorloop =
WordsVEC.begin();
 while (vectorloop != WordsVEC.end())
 {
  cout << *vectorloop << endl; vectorloop++;
 }

 return 0;
}
```

Kao što vidimo, u ovom primeru se prvo kreira mapa pod nazivom WordsMAP čija je inicijalna dužina nula. Pri dodavanju novih elemenata u mapu, automatski se veličina mape proširuje kako bi mogla da prihvati nove elemente. Kretanje kroz mapu, kao što smo spomenuli, se vrši korišćenjem iteratora. U ovom, slučaju, koristimo while petlju koja obezbeđuje kretanje iteratora od prvog do poslednjeg elementa u mapi, i odgovarajuću štampu na ekranu. U drugom delu programa se kreira vektor koji će da služi kao niz stringova. U ovaj vektor zatim ubacujemo 5 elemenata što automatski utiče na prišireni je njegove veličine. Kretanje kroz vektor se takođe vrši korišćenjem iteratora, pomoću koga pristupamo odgovarajućem elementu a zatim ga i štampamo na ekran. Iteratori će naravno biti detaljno opisani u nastavku lekcije.

ADAPTIVNI KONTEJNERI - KONTEJNERI SPECIJALNE NAMENE

Adaptivni kontejneri su predefinisani kontejneri koji su prilagođeni kako bi bili korišćeni za specijalne potrebe. U adaptivne kontejnere spadaju stek, red i prioritetni red

Adaptivni kontejneri su predefinisani kontejneri koji su prilagođeni kako bi bili korišćeni za specijalne potrebe. Zanimljiv deo oko prilagodljivih kontejnera je taj da korisnik može da izabere koji deo kontejnera želi da koristi u određenom trenutku. U kontejnere specijalne namene spadaju:

- Stek - **stack** - je kontejner čiji se elementi procesiraju po LIFO (Last In, First Out – poslednji koji je ušao je prvi koji izlazi) principu, gde se elementi uvek ubacuju (pushed) na kraj odnosno izbacuju (popped) sa kraja kontejnera. Stekovi su obično implementirani korišćenjem dekova (deque) ali mogu biti implementirani i korišćenjem vektora odnosno liste.
- Red - **queue** - je kontejener čiji se elementi procesiraju po FIFO (First In, First Out - prvi koji je ušao je prvi koji izlazi) principu, gde se elementi ubacuju (pushed) na kraj kontejnera ali se brišu (popped) sa početka kontejnera. Redovi se obično implementiraju korišćenjem deka (deque) ali mogu biti implementirani i korišćenjem liste.
- Prioritetni red - **priority queue** - je tip reda gde se elementi čuvaju u sortiranom redosledu (sortiranje korišćenjem operatora <). Kada se neki element ubaci u red automatski se vrši sortiranje elemenata. Iz prioritetnog reda se uvek briše element najvećeg prioriteta a na njegovo mesto dolazi prvi sledeći u hijerarhiji koji je po prioritetu ispod njega.

KREIRANJE KONTEJNERA ZA STEK

Stek je kontejner čiji elementi se procesiraju po LIFO (poslednji koji je ušao je prvi koji izlazi) principu, gde se elementi uvek ubacuju na vrh, odnosno izbacuju sa vrha kontejnera

U nastakvu je prikazan primer u kome se definiše klasa Stack<> za koju su implementirane generičke funkcije pomoću kojih se podaci ubacuju u stek, odnosno izbacuju iz njega:

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

template <class T>
class Stack {
private:
 vector<T> elems;  // elements
public:
 void push(T const&);  // push element
 void pop();       // pop element
 T top() const;    // return top element
 bool empty() const{    // return true if
empty.
  return elems.empty();
 }
};
```

U prethodnom delu koda osim deklaracije šablona navedene su definicije funkcija ubacivanja i izbacivanja podatak iz steka. U nastavku su ostale funkcije i glavni program:

```
template <class T>
T Stack<T>::top () const
{
 if (!elems.empty()) return elems.back();
}

int main()
{
 Stack<int> intStack;  // stack of ints
 Stack<string> stringStack;         // stack of
strings

 intStack.push(7);
 cout << intStack.top() <<endl;

 stringStack.push("hello");
 cout << stringStack.top() << std::endl;
 stringStack.pop();
 stringStack.pop();
}
```

Naravno, ovo je nepotpuna klasa za stek, jer fale izuzeci (sto cemo videcemo u sledecoj lekciji). Rezultat programa biće:

```
7
hello
```

KREIRANJE KONTEJNERA ZA STEK

Stek je kontejner čiji elementi se procesiraju po LIFO (poslednji koji je ušao je prvi koji izlazi) principu, gde se elementi uvek ubacuju na vrh, odnosno izbacuju sa vrha kontejnera

U nastakvu je prikazan primer u kome se definiše klasa Stack<> za koju su implementirane generičke funkcije pomoću kojih se podaci ubacuju u stek, odnosno izbacuju iz njega:

```
template <class T>
void Stack<T>::push (T const& elem)
{
 elems.push_back(elem);
}

template <class T>
void Stack<T>::pop ()
{
 if (!elems.empty()) elems.pop_back();
}
```

U prethodnom delu koda osim deklaracije šablona navedene su definicije funkcija ubacivanja i izbacivanja podatak iz steka. U nastavku su ostale funkcije i glavni program:

```
template <class T>
T Stack<T>::top () const
{
 if (!elems.empty()) return elems.back();
}

int main()
{
 Stack<int> intStack;  // stack of ints
 Stack<string> stringStack;         // stack of
strings

 intStack.push(7);
 cout << intStack.top() <<endl;

 stringStack.push("hello");
 cout << stringStack.top() << std::endl;
 stringStack.pop();
 stringStack.pop();
}
```

Naravno, ovo je nepotpuna klasa za stek, jer fale izuzeci (sto cemo videcemo u sledecoj lekciji). Rezultat programa biće:

```
7
hello
```

STL Iteratori

STL, kontejneri, iteratori, kretanje po kontejneru

- *Uvod u iteratore*
- *Funkcije za rad sa iteratorima*
- *Upotreba iteratora – Kretanje kroz vektor i listu*
- *Upotreba iteratora – Kretanje kroz set*
- *Upotreba iteratora – Kretanje kroz mapu*

07

UVOD U ITERATORE

Iterator je specijalan objekat koji omogućava kretanje po kontejeneru, s tim da korisnik ne mora da poznaje način kako je iterator implementiran

Dizajneri STL biblioteke hteli su da naprave univerzalnu metodu za putovanje (kretanje) kroz sve tipove kontejnera. U tu svrhu, STL definiše iteratore. Iterator je specijalan objekat koji omogućava kretanje po kontejeneru, s tim da korisnik ne mora da poznaje način kako je klasa implementirana. Kod velikog broja kontejnerskih klasa (kao što su liste ili asocijativne klase) iteratori predstavljaju osnovni način pristupa elementima ovih klasa. Iterator je objekat koji pokazuje na članove kontejnera tj. na objekte u kontejneru. Svaki iterator podržava sledeće funkcije:

- Klasa može da vrati iterator koji pokazuje na 1. član kolekcije
- Iterator može da se pomera od jednog do drugog člana
- Program može da učita element pokazan pomoću iteratora

Iterator se može koristiti da se krećete kroz vektor ili listu, i kroz druge tipove kontejnera. Npr. neka se u program kreira lista objekata tipa Student. Program zatim može da koristi iterator pod nazivom „iter“ sa ciljem da se omogući kretanje kroz listu. Deklaracija iteratora bi imala sledeći oblik:

```
list<Student>::iterator
```

Iterator se sastoji iz ogromnog broja predefinisanih funkcija koje olakšavaju kretanje po kontejneru, kao što su na primer:

- **Operator *** - Dereferenciranje interatora koje kao rezultat vraća element na koji iterator trenutno pokazuje.
- **Operator ++** - Pomera iterator na sledeći element kontejnera. Najveći broj iteratora sadrži i preklopljeni operator **–** koji služi za pomeranje na prethodni element.
- **Operator == i Operator !=** - Osnovni operatori poređenja koji ispituju da li dva različita iteratora pokazuju na isti element kontejnera. U slučaju da želimo da poredimo vrednosti elemenata na koje pokazuju iteratori, neophodno je prvo dereferencirati iteratore pa tek onda izvršiti poređenje.
- **Operator =** - Dodeljuje iteratoru novu poziciju (obično je to pozicija elementa na početku ili kraju kontejnera). Naravno, da bi smo dodelili vrednost nekom elementu na koji iterator pokazuje, neophodno je prvo dereferencirati iterator pa tek onda izvršiti operaciju dodele.

FUNKCIJE ZA RAD SA ITERATORIMA

Kod velikog broja kontejnerskih klasa (kao što su liste ili asocijativne klase) iteratori predstavljaju osnovni način pristupa elementima ovih klasa

Svaki kontejner sadrži sledeće četiri osnovne funkcije koje se koriste uz operator =:

- **begin()** - vraća iterator koji pokazuje na prvi element u kontejneru.
- **end()** - vraća iterator koji pokazuje na poziciju koja je prva iza poslednjeg elementa u kontejeneru.
- **cbegin()** – vraća konstantni (dozvoljeno samo čitanje tj. read-only) iterator koji predstavlja početni element u kontejneru.
- **cend()** – vraća konstantni (read-only) iterator koji pokazuje na poziciju koja je prva iza poslednjeg elementa u kontejeneru.

Možda deluje čudno da end() ne pokazuje na poslednji element u listi, ali ovo je urađeno sa ciljem da se ostvari lakoća kretanja u petlji kroz elemente kontejnera, tj. kretanje kroz elemente kontejenera se vrši sve dok se ne dostigne pozicija end(), i tako znamo da smo prošli kroz sve elemente. Konačno, svi kontejneri sadrže bar sledeća dva tipa iteratora:

- **container::iterator** - obezbeđuje iterator za čitanje/pisanje istovremeno
- **container::const_iterator** - obezbeđuje iterator samo za čitanje (read-only)

Iteratori obezbeđuju jedan veoma jednostavan način za putovanje kroz elemente kontejnerske klase bez poznavanja načina kako je neka kontejnerska klasa implementirana. Kada kombinujemo STL algoritme i funkcije članice kontejnerske klase, iteratori postaju još moćniji. U nastavku su dati razni primeri korišćenja iteratora.

UPOTREBA ITERATORA – KRETANJE KROZ VEKTOR I LISTU

Svaki kontejner sadrži sledeće četiri osnovne funkcije koje se koriste zajedno sa iteratorom: begin(), end(), cbegin() i end()

- **Kretanje kroz vektor**

Za početak pogledajmo primer koji demonstrira upotrebu iteratora kod vektora:

```
#include <iostream>
#include <vector>
int main()
{
            using namespace std;

            vector<int> vect;
            for (int nCount=0; nCount < 6; nCount++)
                        vect.push_back(nCount);

            vector<int>::const_iterator it; //
declare an read-only iterator
            it = vect.begin(); // assign it to the
start of the vector
            while (it != vect.end()) // while it
hasn't reach the end
            {
                        cout << *it << " "; // print
value of element it points to
                        it++; // and iterate to the
next element
            }

            cout << endl;
}
```

Rezultat programa biće:

```
0 1 2 3 4 5
```

- **Kretanje kroz listu**

Uradimo sada neke stvari nad listom:

```
#include <iostream>
#include <list>
int main()
{
    using namespace std;

    list<int> li;
    for (int nCount=0; nCount < 6; nCount++)
        li.push_back(nCount);

    list<int>::const_iterator it; // declare an
iterator
    it = li.begin(); // assign it to the start of
the list
    while (it != li.end()) // while it hasn't reach
the end
    {
        cout << *it << " "; // print value of
element it points to
        it++; // and iterate to the next element
    }

    cout << endl;
}
```

Rezultat će ponovo biti:

```
0 1 2 3 4 5
```

Primetimo da je prethodni kod gotovo identičan kao kod primera sa vektorom, iako i vektor i lista imaju kompletno različitu unutrašnju implementaciju!

UPOTREBA ITERATORA – KRETANJE KROZ SET

Kretanje kroz set korišćenjem iteratora je u osnovi identično kao kretanje iteratora kroz listu ili vektor

U narednom primeru, kreiraćemo set koji ima 6 elemenata, i koristi iterator za štampanje vrednosti elemenata seta:

```
#include <iostream>
#include <set>
int main()
{
    using namespace std;

    set<int> myset;
    myset.insert(7);
    myset.insert(2);
    myset.insert(-6);
    myset.insert(8);
    myset.insert(1);
    myset.insert(-4);

    set<int>::const_iterator it; // declare an
iterator
    it = myset.begin(); // assign it to the start of
the set
    while (it != myset.end()) // while it hasn't
reach the end
    {
        cout << *it << " "; // print the value of
the element it points to
        it++; // and iterate to the next element
    }

    cout << endl;
}
```

Rezultat programa će biti:

```
-6 -4 1 2 7 8
```

Primetimo da iako je u ovom primeru sa setovima izvršeno drugačije popunjavanje u odnosu na dodavanje elemenata u listu ili vektor, kretanje kroz set korišćenjem iteratora je u osnovi identično kao kretanje iteratora kroz listu ili vektor.

UPOTREBA ITERATORA – KRETANJE KROZ MAPU

Iterator za mapu je pokazivač na objekat koji sadrži dva elementa. Element first se odnosi na ključ dok se element second odnosi na podatak koji odgovara ključu mape

Kretanje iteratorom kroz mapu je nešto složenije u odnosu na prethodne primere. Mape i muli-mape služe za čuvanje asocijativnog para elemenata (par je definisan kao stl::pair). Ovde koristimo pomoćnu funkciju make_pair() za lakše kreiranje para. std::pair dozvoljava pristup elementima para pomoću funkcija članica first() i second(). U našoj mapi, koristimo first() kao ključ a second() kao vrednost.

```
#include <iostream>
#include <map>
#include <string>
int main()
{
 using namespace std;

 map<int, string> mymap;
 mymap.insert(make_pair(4, "apple"));
 mymap.insert(make_pair(2, "orange"));
 mymap.insert(make_pair(1, "banana"));
 mymap.insert(make_pair(3, "grapes"));
 mymap.insert(make_pair(6, "mango"));
 mymap.insert(make_pair(5, "peach"));
```

Program će proizvesti sledeći rezultat:

```
1=banana 2=orange 3=grapes 4=apple 5=peach 6=mango
```

Primetimo ovde kako je korišćenjem iteratora izvršeno veoma jednostavno kretanje kroz svaki element kontejnera. Stoga korisnik ne mora da vodi računa o načinu kako mapa čuva svoje podatke!

UPOTREBA ITERATORA – KRETANJE KROZ MAPU

Iterator za mapu je pokazivač na objekat koji sadrži dva elementa. Element first se odnosi na ključ dok se element second odnosi na podatak koji odgovara ključu mape

Kretanje iteratorom kroz mapu je nešto složenije u odnosu na prethodne primere. Mape i muli-mape služe za čuvanje asocijativnog para elemenata (par je definisan kao stl::pair). Ovde koristimo pomoćnu funkciju make_pair() za lakše kreiranje para. std::pair dozvoljava pristup elementima para pomoću funkcija članica first() i second(). U našoj mapi, koristimo first() kao ključ a second() kao vrednost.

```
 map<int, string>::const_iterator it; //
declare an iterator
 it = mymap.begin(); // assign it to the start
of the vector
 while (it != mymap.end()) // while it hasn't
reach the end
 {
  cout << it->first << "=" << it->second << "
"; // print the value of the element it points
to
  it++; // and iterate to the next element
 }

 cout << endl;
}
```

Program će proizvesti sledeći rezultat:

```
1=banana 2=orange 3=grapes 4=apple 5=peach 6=mango
```

Primetimo ovde kako je korišćenjem iteratora izvršeno veoma jednostavno kretanje kroz svaki element kontejnera. Stoga korisnik ne mora da vodi računa o načinu kako mapa čuva svoje podatke!

STL Algoritmi

STL biblioteka, algoritmi standardne biblioteke, zaglavlje algorithm

- *Osnovi o STL algoritmima*
- *Funkcije min_element, max_element, find (i list::insert)*
- *Funkcije sort i reverse*

08

OSNOVI O STL ALGORITMIMA

STL pruža veliki broj generičkih algoritama za rad sa elementima kontejnerskih klasa. Ovi algoritmi su implementirani kao globalne funkcije koje koriste iteratore

Kao dodatak na kontejnerske klase i iteratore, STL pruža veliki broj generičkih algoritama za rad sa elementima kontejnerskih klasa. Ovi algoritmi omogućavaju stvari kao što su: sortiranje, pretraga, ubacivanje, promena redosleda, brisanje i kopiranje elemenata kontejnerske klase.

Ovi algoritmi su implementirani kao globalne funkcije koje koriste iteratore. To znači da svaki algoritam treba biti implementiran jednom i automatski će biti upotrebljiv kod svih vrsti kontejnera koje sadrže iteratore (uključujući i korisnički definisanu kontejnersku klasu). Dok je ovo veoma moćan alati i omogućava kreiranje kompleksnog koda veoma brzo, postoji i tamna strana upotrebe ovih algoritama a to je da pojedine kombinacije algoritama i kontejnerskih klasa nisu kompatibilne što može da izazove beskonačnu petlju, pa ih stoga koristite na spostveni rizik. STL pruža svega nekoliko algoritama koje ćemo spomenuti u nastavku. U cilju upotrebe STL algoritama neophodno je da se uključi fajl sa zaglavljima pod nazivom algorithm.

FUNKCIJE MIN_ELEMENT, MAX_ELEMENT, FIND (I LIST::INSERT)

Algoritmi min_element i max_element pronalaze najmanji odnosno najveći element u kontejnerskoj klasi dok funkcija find pronalazi odgovarajuću vrednost u STL listi

Algoritmi min_element i max_element pronalaze najmanji odnosno najveći element u kontejnerskoj klasi:

```
#include <iostream>
#include <list>
#include <algorithm>
int main()
{
            using namespace std;

            list<int> li;
            for (int nCount=0; nCount < 6; nCount++)
                        li.push_back(nCount);

            list<int>::const_iterator it; // declare
an iterator
            it = min_element(li.begin(), li.end());
            cout << *it << " ";
            it = max_element(li.begin(), li.end());
            cout << *it << " ";

            cout << endl;
}
```

Rezultat programa je:

```
0 5
```

U ovom primeru ćemo koristiti find() algoritam da bi smo pronašli neku vrednost u STL listi a zatim ćemo koristiti funkciju list::insert() da bi smo novu vrednost ubacili na tu poziciju u listi.

```
#include <iostream>
#include <list>
#include <algorithm>
int main()  {
 using namespace std;
 list<int> li;
 for (int nCount=0; nCount < 6; nCount++)
  li.push_back(nCount);
 list<int>::const_iterator it;
 // find the value 3 in the list
 it = find(li.begin(), li.end(), 3);
 // use list::insert to insert the value 8
before it
 li.insert(it, 8);
 for (it = li.begin(); it != li.end(); it++)
  cout << *it << " ";
 cout << endl;
```

Na ekranu će biti oštampano:

```
0 1 2 8 3 4 5
```

FUNKCIJE SORT I REVERSE

Funkcija sort se koristi za sortiranje STL kontejnera dok se funkcija reverse koristi za obrtanje redosleda elemenata u kontejneru

U ovom programu izvršićemo sortiranje vektora a zatim ćemo obrnuti redosled elemenata u njemu.

```
#include <iostream>
#include <vector>
#include <algorithm>
int main()
{
 using namespace std;
 vector<int> vect;
 vect.push_back(7);
 vect.push_back(-3);
 vect.push_back(6);
 vect.push_back(2);
 vect.push_back(-5);
 vect.push_back(0);
 vect.push_back(4);

 sort(vect.begin(), vect.end()); // sort the
list
```

Nakon kompajliranja i izvršenja programa biće prikazan sledeći rezultat:

```
-5 -3 0 2 4 6 7
7 6 4 2 0 -3 -5
```

Ovde treba naglasiti da algoritam sort() ne radi sa kontejnerskom klasom za listu — klasa za listu sadrži sopstvenu funkciju članicu koja vrši sortiranje elemenata i koja je mnogo efikasnija od generičke verzije funkcije sort.

FUNKCIJE SORT I REVERSE

Funkcija sort se koristi za sortiranje STL kontejnera dok se funkcija reverse koristi za obrtanje redosleda elemenata u kontejneru

U ovom programu izvršićemo sortiranje vektora a zatim ćemo obrnuti redosled elemenata u njemu.

```
 vector<int>::const_iterator it;
 for (it = vect.begin(); it != vect.end();
it++)
  cout << *it << " ";

 cout << endl;

 reverse(vect.begin(), vect.end()); // reverse
the list

 for (it = vect.begin(); it != vect.end();
it++)
  cout << *it << " ";

 cout << endl;
}
```

Nakon kompajliranja i izvršenja programa biće prikazan sledeći rezultat:

```
-5 -3 0 2 4 6 7
7 6 4 2 0 -3 -5
```

Ovde treba naglasiti da algoritam sort() ne radi sa kontejnerskom klasom za listu — klasa za listu sadrži sopstvenu funkciju članicu koja vrši sortiranje elemenata i koja je mnogo efikasnija od generičke verzije funkcije sort.

STL Stringovi

STL, string klasa, zaglavlje string, funkcije i stringovi

- *Klasa za stringove - std::string*
- *Osnovni pregled string klase*
- *Funkcije za rad sa string klasom*
- *Pristup karakterima stringa*
- *Pretvaranje stringa u C-string (niz karaktera)*
- *Iteratori i stringovi*

09

KLASA ZA STRINGOVE - STD::STRING

Jedna od mana C-stringova je ta da je celokupno upravljanje memorijom ostavljeno korisniku. Standardna C++ biblioteka pruža mnogo bolji način za rad sa stringovima a to je std:string klasa

Standardna biblioteka sadrži mnogo korisnih klasa ali najviše korišćena klasa je verovatno std::string koju smo već pomenuli u okviru C++ kursa, kao i veliki broj njenih funkcija. std::string je klasa koja sadrži veliki broja funkcija za dodelu, poređenje i izmenu stringova. U ovoj sekciji će biti detaljno opisana funkcionalnost string klase kao deo STL biblioteke.

- **Motivacija za upotrebu i kreiranje string klase**

Kao što smo već spomenuli, C-stringovi su predstavljeni kao nizovi karaktera. Primetili smo naravno pri radu sa C-stringovima da nedostatak funkcija otežava njihovu primenu, i veoma lako može doći do grešaka u kodu. Jedna od mana C-stringova je ta da je celokupno upravljanje memorijom ostavljeno korisniku. Na primer, ukoliko želimo da string "hello!" dodelimo baferu neophodno je da prvo izvršimo dinamičko alociranje bafera korišćenjem korektne dužine, na sledeći način:

```
char *strHello = new char[7];
```

Naravno, ne smemo zaboraviti i dodatni '\0' karakter koji treba da dodamo na kraj stringa. Zatim treba iskopirati vrednosti odgovarajućeg teksta u C-string promenljivu:

```
strcpy(strHello, "hello!");
```

Na kraju korišćenja, pošto je string dinamički alociran, neophodno je da dealociramo prostor korišćenjem operatora delete:

```
delete[] strHello;
```

i pritom ne treba zaboraviti da moramo koristiti operator za brisanje dinamičkog niza [] umesto običnog podatka!
Osim toga, ogroman broj funkcija koje C pruža za rad sa brojevima, kao što su dodela vrednosti i poređenje, nije funkcionalno kod C-stringova. Npr, poređenje C-stringova korišćenjem == će ustvari uporediti pokazivače na prvi element niza karaktera dok će korišćenje operatora = ustvari iskopirati adresu jednog pokazivača u drugi. Zbog ovih stvari program može da pukne a greška da bude teško otkrivena čak i u debug modu!.
Na svu sreću, standardna C++ biblioteka pruža mnogo bolji način za rad sa stringovima a to je std::string klasa. Korišćenjem OO C++ koncepata kao što su konstruktori, destruktori, preklapanje operatora, klasa string dozvoljava kreiranje i manipulaciju stingovima na jedan jednostavan, bezbedan i intuitivan način - bez razmišljana o upravljanju memorijom, bez korišćenja složenih naziva funkcija i straha da će nepogodno korišćenje izazvati neočekivan prekid programa.

OSNOVNI PREGLED STRING KLASE

U okviru standardnog fajla "string" C++ standardne biblioteke postoje 3 različite klase za rad sa stringovima i to: basic_string, string i wstring. Najčešće korišćenja klasa je naravno string

Kao što smo ranije spomenuli sva funkcionalnost stringova standardne C++ biblioteke je smeštena u <string> fajlu zaglavlja. U okviru ovog fajla ustvari postoje 3 raličite string klase. Prva je osnovna šablonska klasa pod nazivom basic_string:

```
namespace std
{
          template<class charT,
                    class traits = char_traits<charT>,
                    class Allocator = allocator<charT> >
                    class basic_string;
}
```

a druge dve su ustvari specijalizovane klase osnovne klase basic_string koje ustvari možemo da koristimo kod stringova.

```
namespace std
{
    typedef basic_string<char> string;
    typedef basic_string<wchar_t> wstring;
}
```

U okviru ovog kursa nećemo opisati rad sa klasom wstring, već ćemo se zadržati samo na rad sa string klasom, i samo njena funkcionalnost će biti opisana u nastavku.

FUNKCIJE ZA RAD SA STRING KLASOM

Standardna C++ biblioteka nudi ogroman broj funkcija koje obezbedjuju jednostavnu i sigurnu upotrebu STL string klase

Na sledećim slikama je kompletna lista funkcija koje se koriste nad string klasom.

Funkcija	Efekat
Kreiranje i uništavanje stringova	
(constructor) (destructor)	Kreira ili kopira string uništava string
Veličina i kapacitet	
capacity() empty() length(), size() max_size() reserve()	Vraća broj karaktera koje može string da primi bez realociranja Vraća true ili false u zavisnosti da li je string prazan ili ne Vraća broj karaktera u stringu Vraća maksimalnu dužinu koja može biti alocirana za string Proširuje ili smanjuje kapacitet stringa
Pristup elementima stringa	
[], at()	Pristup karakteru na određenom indeksu
Izmene stringova	
=, assign() +=, append(), push_back() insert() clear() erase() replace() resize() swap()	Dodeljuje novu vrednost stringu Dodaje drugi string na kraj prvog stringa Ubacuje karakter na kraj stringa Ubacuje karakter na proizvoljno mesto u stringu Briše sve karaktere iz stringa Briše karakter sa proizvoljnog mesta iz stringa Zamenjuje karakter na proizvoljnom indeksu u stringu sa drugim karakterom Proširuje ili skraćuje veličinu stringa (odseca ili dodaje karaktere na kraj) Razmenjuje vrednosti dva stringa
Ulaz i izlaz	
>>, getline() << c_str copy() data()	Učitava podatke sa ulaznog toka i smešta u string Štampa string u izlazni tok Pretvara tekst objekta klase string u C-string Vraća sadržaj C++ stringa kao niz karaktera (sa null karakterom) Vraća sadržaj C++ stringa kao niz karaktera (bez null karaktera)

Slika-1 Funkcije za rad sa string klasom – Prvi deo

Na slici 1 su prikazane funkcije za kreiranje i uništavanje stringova, određivanje veličine i kapaciteta, kao i set funkcija za izmenu stringova i ulazno/izlazne operacije. Na narednoj slici 2 su prikazane funkcije za poređenje stringova, spajanje i odsecanje stringova, kao i funkcije za pretragu karaktera u stringu i funkcije za kretanje kroz string pomoću iteratora.

Funkcija	Efekat
Poređenje stringova	
==, != <, <=, > >= compare()	Ispituje da li su stringovi jednaki/različiti (rezultat je bool) Uporedjuje dva stringa (rezultat je tipa bool) Ispituje da li su stringovi jednaki/različiti (rezultat je -1, 0, ili 1)
Podstringovi i spajanje stringova	
+ substr()	Spajanje (sabiranje) dva stringa Vraća podstring stringa (od/do)
Pretraga stringova	
find find_first_of find_first_not_of find_last_of find_last_not_of rfind	Nalazi indeks prvog karaktera / podstringa Nalazi indeks prvog karaktera iz skupa karaktera Nalazi indeks prvog karaktera koji nije iz skupa karaktera Nalazi indeks poslednjeg karaktera iz skupa karaktera Nalazi indeks poslednjeg karaktera koji nije iz skupa karaktera Nalazi indeks poslednjeg karaktera / podstringa
Iterator i podrška alokatora	
begin(), end() get_allocator() rbegin(), rend()	Funkcije koje obezbedjuju rad sa iteratorom (početak/kraj stringa) Vraca alokator Podrška iteratoru za kretanje od kraja ka početku stringa

Slika-2 Funkcije za rad sa string klasom – Drugi deo

PRISTUP KARAKTERIMA STRINGA

Najlakši način pristupa karakterima stringa je korišćenjem preklopljene verzije operatora indeksiranja [ ]. Drugi način je korišćenje funkcije članice: at(indeks)

Postoje dva skoro identična načina pristupa karakterima C++ stringa. Lakši i brži način je korišćenjem preklopljene verzije operatora indeksiranja []. Zaglavlja ovih preklopljenih funkcija su:

```
char& string::operator[] (size_type nIndex)
const char& string::operator[] (size_type nIndex) const
```

- Obe funkcije kao rezultat vraćaju karakter koji se nalazi na indeksu nIndex.
- Ukoliko prosledite neki neodgovarajući indeks ponašanje će biti potpuno neodređeno.
- Korišćenje funkcije length() kao indeks je validno ali samo za konstantne stringove. Ova funkcija vraća vrednost koju generiše podrazumevajući konstruktor string klase. Ipak nije baš poželjno koristiti ovaj slučaj.
- S obzirom da je povratni tip char&, moguće je koristiti ovaj preklopljeni operator za izmene elemenata stringa na određenom indeksu.

Jedan primer upotrebe je:

```
string sSource("abcdefg");
cout << sSource[5] << endl;
sSource[5] = 'X';
cout << sSource << endl;
```

pri čemu će izlaz programa biti:

```
f
abcdeXg
```

Postoji i drugi način. Ova druga verzija je sporija jer koristi izuzetke da proveri da li je nIndex validan indeks. Ukoliko niste sigurni da li je nIndex validan ili ne, onda je pogodnije koristiti ovu verziju da bi ste pristupili stringu. Postoje dva različita zaglavlja i to:

```
char& string::at (size_type nIndex)
const char& string::at (size_type nIndex) const
```

- Obe funkcije kao rezultat vraćaju karakter koji se nalazi na indeksu nIndex.
- Ukoliko prosledite neodgovarajući indeks izbaciće se izuzetak tipa out_of_range.
- I u ovom slučaju, s obzirom da je povratni tip char&, moguće je koristiti ovaj preklopljeni operator za izmene elemenata stringa na određenom indeksu.

Primer upotrebe je :

```
string sSource("abcdefg");
cout << sSource.at(5) << endl;
sSource.at(5) = 'X';
cout << sSource << endl;
```

pri čemu će izlaz ponovo biti:

```
f
abcdeXg
```

PRETVARANJE STRINGA U C-STRING (NIZ KARAKTERA)

Osnovne funkcije koje se koriste za pretvaranje teksta string objekta u C-string su: c_str (), data() i copy()

Veliki deo funkcija (uključujući sve funkcije C biblioteke) očekuju da stringovi budu formatirani u C-stilu a na u formatu C++ stingova. Iz ovog razloga, std::string pruža tri različita načina za konvertovanje objekata string klase u C-stringove.

Prvi oblik je funkcija **c_str ()** sa zaglavljem:

```
const char* string::c_str () const
```

- Vraća pokazivač na C string za uneti objekat string klase.
- Na kraj C stringa dodaje null karakter kraja C stringa.
- Dobijeni string je deo objekta C++ string klase i stoga ne sme biti obrisan

Primer upotrebe je:

```
string sSource("abcdefg");
cout << strlen(sSource.c_str());
```

pri čemu će izlaz biti:

```
7
```

Drugi oblik je funkcija data() sa zaglavljem

```
const char* string::data () const
```

- Vraća sadržaj objekta string klase kao konstantan C – string.
- Pritom ne dodaje nul karakter na kraj C-stringa
- Takođe, dobijeni C-string je deo objekta C++ string klase i stoga ne sme biti obrisan

Primer upotrebe je:

```
string sSource("abcdefg");
char *szString = "abcdefg";
// memcmp compares the first n characters of two
// C-style strings and returns 0 if they are
equal
if (memcmp(sSource.data(), szString,
sSource.length()) == 0)
                    cout << "The strings are
equal";
else
                    cout << "The strings are not
equal";
```

Izlaz će biti:

```
The strings are equal
```

Treći oblik je funkcija copy() sa sledećim zaglavljem:

```
size_type string::copy(char *szBuf, size_type nLength)
const
size_type string::copy(char *szBuf, size_type nLength,
size_type nIndex) const
```

- Obe verzije funkcije kopiraju najviše nLength stinga u szBuf, počevši od karaktera sa indeksom nIndex.
- Rezultat funkcije je broj iskopiranih karaktera.
- Ne dodaje se nul karakter na kraj stringa. Korisnik je taj koji treba da obezbedi da je szBuf inicijalizovan sa NULL i da na kraj stringa, preko vraćenog length doda nul karakter
- Korisnik je taj koji mora da vodi računa da se ne prepuni szBuf.

Primer upotrebe je:

```
string sSource("sphinx of black quartz,
judge my vow");
char szBuf[20];
int nLength = sSource.copy(szBuf, 5, 10);
szBuf[nLength]='\0';  // Make sure we
terminate the string in the buffer
cout << szBuf << endl;
```

Izlaz će biti: black.

Treba napomenuti da je za potrebe ovog kursa dovoljno poznavati funkciju c_str() pošto je ona najjednostavnija i najsigurnija od sve tri pomenute funkcije.

ITERATORI I STRINGOVI

Klasa string sadrži iterator koji omogućava kretanje kroz karaktere stringa. Iterator stringa je tipa: string::const_iterator

Klasa string sadrži iterator koji omogućava kretanje kroz karaktere stringa. U nastavku je prikazan demonstrativan primer upotrebe iteratora kod stringova:

```
// Using an iterator to output a string.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
            string string1( "Testing iterators" );
            string::const_iterator iterator1 =
string1.begin();

            cout << "string1 = " << string1
                        << "\n(Using iterator
iterator1) string1 is: ";

            // iterate through string
            while ( iterator1 != string1.end() )
            {
                        cout << *iterator1; //
dereference iterator to get char
                        iterator1++; // advance
iterator to next char
            } // end while

            cout << endl;
            return 0;
} // end main
```

Itetarori omoguavaju pristup svakom pojedinačnom karakteru stringa pomoću sintakse koja je slična ka kod pokazivačke aritmetike.

Na početku main programa se deklariše string promenljiva string1 i iterator iterator1 koji je tipa **string::const_iterator**. const_iterator je ustvari iterator koji ne može da menja karaktere u stringu, tj u ovom slučaju karaktere objekta kroz koji se kreće, tj. string1. Iterator iterator1 se inicijalno postavlja na početak stringa string1 pomoću funkcije članice **begin()**. U klasi string postoje dve verzije funkcije begin, jedna koja vraća iterator za kretanje kroz promenljiv (nekonstantan) sting i druga verzija koja je konstantna i vraća iterator tipa const_iterator za kretanje kroz konstantan string. Ostatak koda je naravno poznat i njega smo opisali u sekciji o iteratorima.

Vežbe

C++ biblioteka, STL, kontejneri, iteratori

- *Upotreba vektora – Primer1*
- *Upotreba vektora – Primer2*
- *Primer. Osnovni algoritmi za pretraživanje i sortiranje*

10

UPOTREBA VEKTORA – PRIMER1

Vektori u STL biblioteci se odnose na dinamičke nizove koji su sposobni da po potrebi rastu koliko je potrebno da bi se u njih ubacili dodatni elementi. Definicija je u fajlu zaglavlja: vector

Posmatrajmo sledeći primer:

```
#include <iostream>
#include <stdlib.h>
#include <vector>
#include <string>
using namespace std;

int main(int argc, char *argv[])
{
          vector<string> namesV;
          namesV.push_back("Toma");
          namesV.push_back("Dana");
          namesV.push_back("Hana");
          namesV.push_back("Ana");
          namesV.push_back("Maja");
          namesV.push_back("Julija");
          cout << namesV[0] << endl;
          cout << namesV[5] << endl;

          return 0;
}
```

Kao što vidimo, ovde koristimo vektore, ali ovde je vektor šablon čiji je šablonski parametar string:

```
vector <string> names;
```

Pri korišćenju vektora obično se mogu primeniti sledeće operacije:

- Dodavanje elemenata na kraj vektora
- Pristup elementima vektora
- Kretanje (korišćenjem iteratora) kroz vektor ili od početka prema kraju ili od kraja prema početku vektora.

UPOTREBA VEKTORA – PRIMER2

Pri korišćenju vektora obično se mogu primeniti operacije kao što su: dodavanje elemenata na kraj, pristup elementima vektora i dvosmerno kretanje kroz vektor korišćenjem iteratora

Pogledajmo sada sledeći primer koji demonstrira upotrebu STL vektora koji je veoma sličan nizu uz dodatak da on automatski savladava nove zahteve skladištenja kada vektor počinje da raste:

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

int main()
{
 vector<int> vec;
 int i;

 cout << "vector size = "
  << vec.size() << endl;

 for(i = 0; i < 5; i++)
 {
  vec.push_back(i);
 }

 cout << "extended vector size = "
  << vec.size() << endl;
```

Nakon kompajliranja i izvršavanja prethodnog programa dobija se sledeći rezultat:

```
vector size = 0
extended vector size = 5
value of vec [0] = 0
value of vec [1] = 1
value of vec [2] = 2
value of vec [3] = 3
value of vec [4] = 4
value of v = 0
value of v = 1
value of v = 2
```

Sledeće funkcije su od značaj kada je u pitanju ovaj primer:

- Funkcija push_back( ) ubacuje vrednost na kraj vektora, i uvećava veličinu vektora ukoliko je neophodno.
- Funkcija size( ) vraća veličinu vektora.
- Funkcija begin( ) vraća iterator na početak vektora.
- Funkcija end( ) vraća iterator na prvu slobodnu poziciju nakon poslednjeg elementa vektora.

UPOTREBA VEKTORA – PRIMER2

Pri korišćenju vektora obično se mogu primeniti operacije kao što su: dodavanje elemenata na kraj, pristup elementima vektora i dvosmerno kretanje kroz vektor korišćenjem iteratora

Pogledajmo sada sledeći primer koji demonstrira upotrebu STL vektora koji je veoma sličan nizu uz dodatak da on automatski savladava nove zahteve skladištenja kada vektor počinje da raste:

```
 cout << "extended vector size = "
  << vec.size() << endl;

 for(i = 0; i < 5; i++)
 {
  cout << "value of vec [" << i << "] = "
   << vec[i] << endl;
 }

 vector<int>::iterator v = vec.begin();
 while( v != vec.end())
 {
  cout << "value of v = " << *v << endl;
  v++;
 }
 return 0;
}
```

Nakon kompajliranja i izvršavanja prethodnog programa dobija se sledeći rezultat:

```
vector size = 0
extended vector size = 5
value of vec [0] = 0
value of vec [1] = 1
value of vec [2] = 2
value of vec [3] = 3
value of vec [4] = 4
value of v = 0
value of v = 1
value of v = 2
```

Sledeće funkcije su od značaj kada je u pitanju ovaj primer:

- Funkcija push_back( ) ubacuje vrednost na kraj vektora, i uvećava veličinu vektora ukoliko je neophodno.
- Funkcija size( ) vraća veličinu vektora.
- Funkcija begin( ) vraća iterator na početak vektora.
- Funkcija end( ) vraća iterator na prvu slobodnu poziciju nakon poslednjeg elementa vektora.

PRIMER. UPOTREBA REDA I REDA SA PRIORITETOM

Red - queue - je kontejener čiji se elementi procesiraju po FIFO principu, dok je prioritetni red - priority queue - tip reda gde se elementi čuvaju u sortiranom redosledu

a) Upotreba reda:

```
#include <iostream>
using std::cout;
using std::endl;
#include <queue> // queue adapter definition
int main()
{
            std::queue< double > values; // queue
with doubles

            // push elements onto queue values
            values.push( 3.2 );
            values.push( 9.8 );
            values.push( 5.4 );

            cout << "Popping from values: ";

            // pop elements from queue
            while ( !values.empty() )
            {
                        cout << values.front() << '
'; // view front element
                        values.pop(); // remove
element
            } // end while

            cout << endl;
            return 0;
} // end main
```

Rezultat će biti:

```
Popping from values: 3.2 9.8 5.4
```

b) Upotreba reda sa prioritetom:

```
#include <iostream>
using std::cout;
using std::endl;
#include <queue> // priority_queue adapter
definition
int main()
{
            std::priority_queue< double >
priorities; // create priority_queue

            // push elements onto priorities
            priorities.push( 3.2 );
            priorities.push( 9.8 );
            priorities.push( 5.4 );

            cout << "Popping from priorities: ";

            // pop element from priority_queue
            while ( !priorities.empty() )
            {
                        cout << priorities.top() <<
' '; // view top element
                        priorities.pop(); // remove
top element
            } // end while

            cout << endl;
            return 0;
} // end main
```

Rezultat je:

```
Popping from priorities: 9.8 5.4 3.2
```

Zadaci za samostalan rad

C++ biblioteka, STL, kontejneri, iteratori

- *Zadaci za samostalno vežbanje*

11

ZADACI ZA SAMOSTALNO VEŽBANJE

Na osnovu materijala za ovu nedelju uraditi samostalno sledeće zadatke:

1. Deklaracija, incijalizacija i rad sa vektorima:

a) Deklarisati jedan vektor sa 5 realnih brojeva, i incijalizovati ih slučajnim (random) vrednostima od 1.0 do 5.0. Koristiti interval od 1 do 500 za generisanje realnog broja zapisanog na dve decimale (npr. slučajan broj 323 će ustvari biti 3.23). Dodati izvorni kod da radi sledeće: a1) zameniti poslednji element sa 25.0; a2) pomnožiti treći element sa 6.0; a3) zadati prvom elementu vrednost 10.0. Prikazati na ekranu svaki element vektora.

b) Deklarisati jedan vektor za slovne promenljive od 8 slova, inicijalizovan sa tekstualnim nizom ‘my_name’, i sa nultim znakom na kraju. Zatim koristiti samo ime vektora da se prikaže taj vektor.

c) Deklarisati 2 vektora sa 3 celobrojne promenljive i inicijalizuj prvi vektor sa vrednostima od 1 do 3. Napisati kod koji će da popuni drugi vektor sa kvadratima elemenata prvog vektora.

2. Napisati program koji traži od korisnika da unese deset brojeva koji će biti zapamćeni u vektoru. Nakon unosa program treba da sortira unete brojeve po veličini, od najvećeg ka najmanjem, i da ih prikaže u konzoli.

3. Napisati C++ program koji formira listu i ispisuje sadržaj u inverznom poretku. Zatim učitati niz karaktera u jednoj liniji (dok se ne pređe u novi red), i korišćenjem steka ispisati karaktere u inverznom poretku.

4. Napisati program koji učitava seriju imena studenata i eleminiše duplikate tako sto pakuje imena u set. Omogućiti korisniku da pretražuje da li se neko ime nalazi u setu.

5. Napisati program koji svaki znak * u učitanom C++ stringu zamenjuje stringom “+“, i prikazuje dobijeni rezultat. Koristiti iteratore za putovanje kroz string.

6. Napisati program koji učitava reči iz tekstualnog fajla i prikazuje na ekran samo one reči koje nemaju duplikate, i to u rastućem alfabetskom poretku. Ime fajla se prosleđuje kao argument komandne linije. Čuvati reči u odgovarajućoj STL kontejnerskoj klasi.

Zaključak

13

REZIME

Na osnovu svega obrađenog možemo zaključiti sledeće:

Svojstva C ++ standardne biblioteke su smeštena u imenskom prostoru std. C++ standardna biblioteka se može podeliti u dva dela: Standardna biblioteka funkcija i Biblioteka OO klasa. Standarna C++ biblioteka sadrži sve elemente C biblioteke, uz male izmene koje obezbeđuju sigurnost upotrebe tipova podataka (type safety). Standardna C++ OO biblioteka definiše dodatni skup klasa koje pružaju podršku mnogobrojnim svakodnevnim aktivnostima, uključujući ulaz/izlaz, rad sa stringovima i procesiranje brojeve.

Kontejnerska klasa je klasa dizajnirana za držanje i organizovanje više instanci druge klase. Obično se javlja u dva oblika i to kao vrednosni odnosno referencni kontejneri. Iako C ++ ima ugrađenu funkcionalnost za rad sa nizovima, programeri će često koristiti kontejnersku klasu za niz umesto standarnog niza. Jednom ispravno napisana kontejnerska klasa može biti korišćena koliko god puta je to neophodno bez potreba za dodatnim izmenama u samoj klasi.

STL kontejnerske klase su komponente STL biblioteke koje se najčešće koriste u praksi. Postoje tri kategorije kontejnerskih klasa i to: sekvencionalni, asocijativni i adaptivni kontejneri. STL ima 3 sekvencionalna kontejnera i to: vektor, dek i listu. Asocijativni kontejneri obezbeđuju direktan pristup svojim elementima preko ključa (search key). Svaki asocijativni kontejner čuva svoje ključeve u sortiranom poretku. U asocijativne kontejnere spadaju skupovi (set), multi-skupovi, mape i multimape. Adaptivni kontejneri su predefinisani kontejneri koji su prilagođeni kako bi bili korišćeni za specijalne potrebe. U adaptivne kontejnere spadaju stekovi, redovi i redovi sa prioritetom.

Iterator je specijalan objekat koji omogućava kretanje po kontejeneru, s tim da korisnik ne mora da poznaje način kako je iterator implementiran. Kod velikog broja kontejnerskih klasa (kao što su liste ili asocijativne klase) iteratori predstavljaju jedini način pristupa elementima ovih klasa. Svaki kontejner sadrži sledeće četiri osnovne funkcije koje se koriste zajedno sa iteratorom: begin(), end(), cbegin() i end().

STL pruža veliki broj generičkih algoritama za rad sa elementima kontejnerskih klasa. Ovi algoritmi su implementirani kao globalne funkcije koje koriste iteratore. Standardna biblioteka sadrži mnogo korisnih klasa ali najviše korišćena klasa je verovatno std::string.