Pokazivači

Ciljevi:

- upoznati se sa pojmom pokazivača
- razumjeti način rada s pokazivačima
- shvatiti razloge uvođenja i korištenja pokazivača

Pregled lekcije

Definicija pokazivača

Pokazivač je varijabla koja sadrži memorijsku adresu neke druge varijable. Pomoću pokazivača možemo pristupiti, preko operatora dereferencijacije (*), i do sadržaja ali i do memorijske adrese gdje je taj sadržaj spremljen. Razlozi zašto su pokazivači toliko značajni biti će uskoro razjašnjeni.

Sintaksa pokazivača:

```
tip podataka *ime;
```

Tip podataka kod pokazivača je potreban kako bi se znalo koliko je velika varijabla u memorijskom prostoru na koji pokazivač pokazuje, tj.,koliko se memorije treba alocirati, ali i zbog toga da bi se znalo na koji se način interpretira podatak smješten u toj memoriji, tj. kojeg je taj podatak tipa podataka.

Isto tako još jedan vrlo važan operator adrese &, koji u ovom kontekstu vraća memorijsku lokaciju neke varijable, tj. gdje je varijabla mještena u memoriji. Primjer djelovanja ovog operatora:

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

int main()
{
    int i = 10;
    int *p = &i;

    cout << ++*p << endl;

    system("pause");

    return 0;
}
```

Ako pak želimo da pokazivač pokazuje na neki konstantan objekt vrijednost mora biti inicijalizirana odmah pri stvaranju pokazivača u suprotnom će doći do greške.

```
const int *p = new const int(777);
```

Napomena: na nepromjenjive objekte se smiju preusmjeravati samo pokazivači koji su deklarirani da pokazuju na nepromjenjive objekte u protivnom će doći do greške pošto bi se mogla narušiti konstantnost nepromjenjivog objekta. Međutim pokazivači na nepromjenjive objekte se smiju preusmjeravati na promjenjive objekte ali u tom slučaju je promjena vrijednosti moguća samo direktno ne i preko pokazivača.

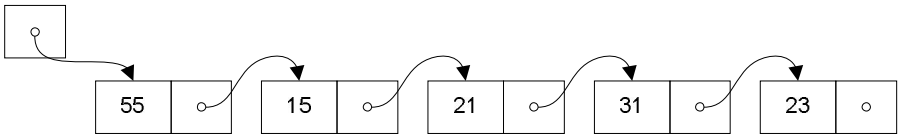
Sada kada smo se upoznali sa pokazivačima pokažimo na sljedećem primjeru jedan vrlo koristan način rada sa pokazivačima.

Primjer 1:
ULAZ: Unijeti N prirodnih brojeva, gdje N nije unaprijed poznat niti se može ograničiti.
Prirodni brojevi se unose sve dok se ne unese broj 0.
IZLAZ: Ispisati brojeve obrnutim redoslijedom.

Ovdje bismo mogli kreirati polje, no time bismo morali zanemariti zahtjev da N ne može biti unaprijed ograničen odozgo, stoga ono što je ideja ovdje je da se ne alocira memorija za sve elemente unaprijed, već da se memorija alocira točno kada zatreba i tu će nam uvelike pomoći pokazivači pošto je upravo pokazivač mehanizam koji omogućuje da se tokom rada programa prema potrebi alocira memorijski prostor.

Ono što ovdje želimo učiniti je da se za svaki broj koji je unesen posebno alocira memorijski prostor ali na taj način ne možemo garantirati da će uzastopni elementi biti na uzastopnim memorijskim lokacijama te stoga elemente moramo međusobno povezati tako da svaki element ima vezu prema sljedećem unesenom elementu. Dakle za povezivanje elemenata nam trebaju pokazivači.
Pretpostavimo da imamo unesene sljedeće brojeve: 23, 31, 21, 15, 55.

Tada ćemo napraviti sljedeću strukturu:



Slika 1: Vezana lista

Ono što se, promatrajući prethodnu sliku i do sada rečeno, može naslutiti je to da svaki element treba biti slog koji će, osim unesene vrijednosti sadržavati i pokazivač na sljedeći uneseni element.

Strukture koje se generiraju pomoću pokazivača i alociranja memorije koje oni omogućuju nazivaju se dinamičke strukture a struktura koju smo opisali malo prije naziva se vezana lista.

Pojam dinamičnosti ove strukture ima dvostruko značenje: strauktura je dinamika jer njena veličina raste i opada s obzirom na potrebe, ali isto tako zbog toga jer se način na koji se memorijski prostor za njene elemente alocira (traži od operacijskog sustava) zove dinamička alokacija memorije. Način na koji se radi s memorijskim prostorom koji je dinamički alociran u programskom jeziku u potpunosti je drugačiji od onoga kako program tretira statički alocirane objekte. Dok se sa statički alociranim objekrima radi pomoću programskog stoga (stack), dinamički alocirani objekti se čuvaju u internoj strukturi koja se naziva programska hrpa (heap).

Slog koji nama treba kako bismo riješili gornji problem izgleda ovako:

```
struct element {
        int vrijednost;
        element *sljedeci;
};
typedef element *lista;
```

Naredbom **typedef** definiramo novi tip podataka lista, koji je pokazivač na strukturu tipa element.

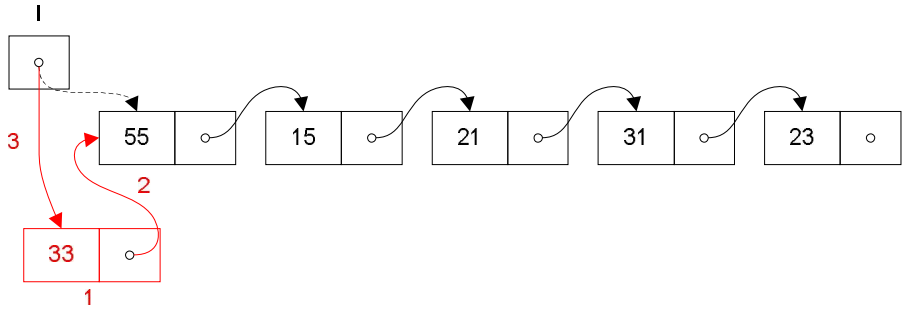
Dakle, ono što mi trebamo napraviti je učitati određene brojeve u i nakon toga ih treba moći ispisati obrnutim redoslijedom.

1. Radi
2. Učitaj prirodni broj
3. Ako je broj > 0
4. Alociraj prostor za novi element liste
5. Upiši vrijednost u novi element liste
6. Poveži novi element na početak liste
7. inače
8. end = true
9. Dok još postoji elemenata u listi
10. Ispiši vrijednost
11. Izbaci element iz liste

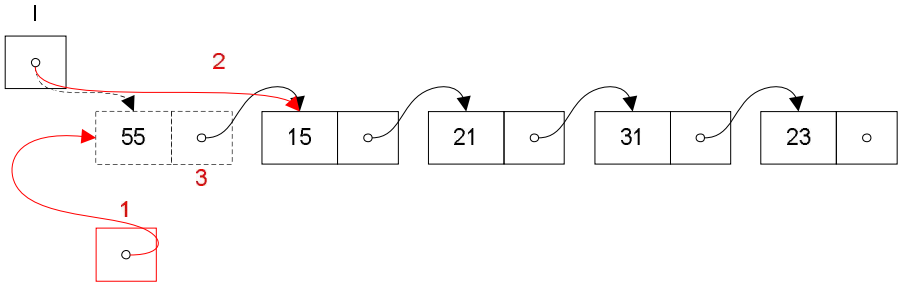
Prije no što iznesemo programski kod rješenja ovog problema, osvrnimo se na dvije operacije na vezanoj listi koje trebamo implementirati – dodavanje novog elementa na početak vezane liste i brisanje elementa s početka vezane liste.

Jasno je da kod dodavanja novog elementa u vezanu listu trebamo alocirati memorijski prostor na kojem će se ovaj element nalaziti. Treba napomenuti da taj memorijski prostor nije ni u kakvoj vezi s memorijskim prostorom koji je dodijeljen pri alokaciji prethodno dodanih elemenata liste. Upravo zbog toga je potrebno da elemente vezane liste međusobno povezujemo pokazivačima. Jasno je da će pookazivač `l` nakon dodavanja novog elementa morati pokazivati (sadržavati memorijsku adresu) upravo dodanog elementa. No, prije no što mu pridružimo adresu novog elementa, potrebno je povezati novi element s ostatkom liste. Ako to ne bismo učinili, ostatak bi liste ostao izgubljen i više ne bismo znali gdje se on u memoriji nalazi. To ćemo učiniti tako da vrijednost pokazivača `l` zapišemo u

pokazivač u slogu novododanog elementa liste. Na taj će način pokazivač iz novododanog elementa pokazivati na sljedeći element vezane liste. Tek nakon što to učinimo možemo pokazivač `l` preusmjeriti tako da pokazuje na novododani element.



Kod brisanja elementa s početka liste pokazivač `l` treba preusmjeriti tako da pokazuje na drugi element u vezanoj listi. No i pri ovoj operaciji, kao i pri dodavanju elemenata na redosiljed kojim ćemo to učiniti. Naime, ako jednostavno preusmjerimo pojkazivač l na drugi element u vezanoj listi, onda će se izguiti adresa elementa koji smo upravo izbacili. Naravno, ovaj nam element više nije potreban, pa tto, što se tiče našeg programa, nije važno. No, memorijski prostor koji taj element zauzima, još uvijek ostaje dodjeljen našem programu, iako se više ne koristi. Takav odnos prema memorijskom prostoru uzrokovat će brzo zauzimanje sve raspoložive memorije. Stoga je iznimno važno da se svaki memorijski prostor koji je dinamički alociran nakon korištenja vrati operacijskom sustavu, odnosno dealocira. Zbog toga se prije no što se pokazivač `l` preusmjeri, njegova vrijednost, odnosno adresa prvog elementa u vezanoj listi mora zapisati u pomoćni pokazivač, kako bi se nakon premještanja pokazivača `l` na sljedeći element liste prostor koji je obrisani element zausimao mogao dealocirati.



Sada kada smo razradili pseudokod možemo početi programirati C++ kod. Pa pogledajmo na koji način bismo implementirali ovaj primjer.

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

struct element {
        int vrijednost;
        element *sljedeci;
};
typedef element *lista;
```

```
int main () {
    lista l = NULL;
    bool end = false;
    element *el;


    do {
       int A;
       do {
          cout << "A = ";
          cin >> A;
       } while (A < 0);
       if (A > 0) {
          el = new element;
          (*el).vrijednost = A;
          (*el).sljedeci = l;
          l = el;
       }
       else end = true;
    } while (!end);

    while (l != NULL) {
          cout << (*l).vrijednost << " ";
          el = l;
          l = (*l).sljedeci;
          delete el;
    }

    cout << endl;
    system("pause");

    return 0;
}
```

NUL-POKAZIVAČ

ALOCIRANJE NOVOG MEMORIJSKOG PROSTORA ZA VARIJABLU TIPA ELEMENT TE PRIDRUŽIVANJE MEMORIJSKE ADRESE POKAZIVAČU EL

NAREDBA DEALOKACIJE MEMORIJSKOG PROSTORA NA KOJI POKAZUJE POKAZIVAČ

Za razliku od ostalih varijabli, za pokazivače možemo definirati da ne sadrže nikakvu memorijsku adresu. Nepostojeća se memorijska adresa definira pomoću konstante NULL. NULL je konstanta iz biblioteke cstdlib, koji predstavlja naziv za broj 0. Jasno je da će uvijek posljednji element u vezanoj listi, odnosno onaj koji je prvi dodan sadržavati NULL-pokazivač.

Nadalje, koristi se naredba **new**, kojom se dinamički alocira prostor. Vidi se da se ova alokacija vrši navodeći tip podataka za koji se dani memorijski prostor alocira, tj. u našem slučaju slog `element`.

Nadalje, koristi se i naredba delete, kojom se dealocira memorijski prostor na koji pokazuje određeni pokazivač, u našem slučaju pokazivač `el`.

Poledajmo sada kakko se koriste pokazivačke varijable. Da bismo pristupili memorijskom prostoru na koji pokazuje neki pokazivač koristimo tzv. već prethodno spomenuti operator dereferencijacije *. Primijetimo da se u izrazu '`(*el).vrijednost = A;`' *el nalazi u zagradama. To je zato što se C++ izrazi interpretiraju zdesna na lijevo, pa bi bez zagrade

operator dereferencijacije značio podatak na koji pokazuje pokazivač el.vrijednost a mi želimo raditi sa našim pokazivačem `el`.

Prije no što krenemo dalje, potrebno je provjeriti radi li naš program dobro u rubnim slučajevima, odnosno kad dodajemo element u praznu listu, odnosno kada brišemo posljednji element iz nje.

Prije no što dodamo prvi element u vezanu listu pokazivač `l` će biti NULL-pokazivač. Slijedimo li operacije dodavanja elementa, vidjet ćemo da će se u pokazivač u slogu novododanog elementa upisti vrijednost pokazivača `l`, odnosno NULL-vrijednost, nakon čega će se pokazivač `l` preusmjeriti tako da pokazuje na novododani element. Kako će prvi dodani element biti posljednji u listi to j upravo ono što želimo učiniti.

S druge strane, ako brišemo posljednji element iz liste, njegov će se pokazivač prepisati u pokazivač `l`. Kako je on posljednji u vezanoj listi, on će sadržavati NULL-pokazivač. Dakle, nakon toga će `l` biti NULL-pokazivač, upravo kako smo i inicijalizirali praznu vezanu listu na početku.

Polja i pokazivači

Do sada smo duljinu polja uvijek definirali fiksno, unaprijed. Pokazivači nam omogućuju da duljinu polja definiramo ovisno o parametru u programu tako da alociramo točno onoliko memorijskog prostora koliko nam je potrebno. Prije no što pokažemo kako se to radi, pogledajmo još jedan jednostavni primjer, koji će pokazati u kakvoj su vezi polja i pokazivači.

Primjer 2:

ULAZ: N cijelih brojeva, N<=50.
IZLAZ: Poredati brojeve tako da na početku polja dolaze neparni brojevi poredani uzlazno, a nakon toga parni brojevi poredani silazno.

Kako bismo ovo postigli radit ćemo dvostruko sortiranje umetanjem, tako da ćemo neparne brojeve umetati s početka polja, a parne s kraja. Imat ćemo dva brojača (i,j). Brojač i će služiti da bi se znalo gdje je kraj sortiranog dijela neparnih brojeva, a j da bi se znalo gdje je početak sortiranog dijela parnih brojeva. Ovi će se kursori pomicati jedan prema drugome sve dok i ne premaši j.

1. Učitati n
2. Učitati n brojeva
3. i=0, j = n-1, b = 0
4. Sve dok je b<=j radi
5. Ako je a[i] neparan, onda ga umetni u sortirani dio polja na početku polja
6. inače ga umetni u sortirani dio na kraju polja
7. ispiši polje

U ovom ćemo primjeru pokazati drugačiji način korištenja polja. Recimo da je deklarirano polje

```
int a[50];
```

Znamo da je a[0] 0-ti element tog polja, međutim, što je samo a? Samo a je zapravo pokazivač na 0-ti element polja. Stoga sada znamo da su polja u biti implementirana pomoću pokazivača što nam otvara razne nove mogućnosti rada s podacima.

Dakle, `a[0]` je isto što i `*a`.

Što se pak tiče same aritmetike s pokazivačima. Pokazivaču je moguće oduzeti ili dodati cijeli broj, dakle dozvoljeni su jedino operatoru + i -. Nadalje u aritmetičkom izrazu smije biti samo jedna pokazivačka varijabla.

Pretpostavimo recimo da je `a = 0x1000`. Postavlja se pitanje koliko će recimo biti `a + 1`? `a + 1` će zapravo biti `0x1004`. Dali ste naslutili možda zašto? Iz tog razloga što se pokazivaču ne dodaje konkretna vrijednost 1 nego mu se dodaje jedan **integer**. Dakle, ako je a pokazicač na nulti element polja, a + 1 će biti pokazivač na prvi element polja. Slično, a + 2 će biti pokazivač na drugi element polja itd.
Pogledajmo C++ kod za ovaj primjer.

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

int main () {

    int n, a[50];
    // unos broja elemenata
    do {
       cout << "N = ";
       cin >> n;
    } while (n<1 || n>50);

    // unos vrijednosti elemenata u polje
    for (int i = 0; i < n; i++){
        cout << "A[" << i << "] = ";
        cin >> *(a+i);
        }

    int i=0, j = n-1, b = 0;
    // sortiranje
    while (b<=j) {

          // ako je broj neparan
         if (*(a+b)%2) {

            int pom = *(a+b);
            int l = i-1;

            while (l>=0 && *(a+l) > pom)    {
            *(a+l+1) = *(a+l--);
            *(a+l+1) = pom;
            }
```

```
            i++;
        }

        // ako je broj paran
         else if(!(*(a+b)%2)) {

            int pom = *(a+b);
            int l = j, tB=b;

            while (tB<l) {
            *(a+tB) = *(a+tB+1);
            *(a+tB+1) = pom;
            tB++;
            }

            if(l<n-1)l++;

            while (l<=n-1 && *(a+l) > pom) {
            *(a+l-1) = *(a+l++);
            *(a+l-1) = pom;
            }

            j--;
            b--;
         }

        b++;
    }

    for (i = 0; i < n; i++) {
    cout << *(a+i) << " ";
    }

    cout << endl;
    system("pause");

    return 0;
}
```

Sada kada smo se bolje upoznali sa pokazivačima i uvidjeli vezu između pokazivača i polja sagledajmo i sljedeći primjer. Pretpostavimo da želimo da korisnik unosi brojeve u intervalu [0, 100] i da je nakon unosa brojeve potrebno poredati po veličini.

Ako je interval iz kojeg elementi poprimaju vrijednosti relativno mali, onda se može koristiti jednostavan ali efikasan algoritam sortiranja. Generira se pomoćno polje tipa koje ima onoliko elemenata koliko ima različitih vrijednosti koje elementi glavnog polja mogu poprimiti. Svi se elementi pomoćnog polja iniciraju na 0. Prođe se jednom po glavnom polju i ako je vrijednost i-tog elementa glavnog polja jednaka j, onda se j-ti element pomoćnog polja poveća za 1. Na kraju će i-ti element pomoćnog polja sadržavati broj pojava vrijednosti i u glavnom polju.

Primjer 3:

ULAZ: N brojeva. Svaki broj mora biti između 0 i 100.

IZLAZ: Poredati brojeve po veličini.

8. Učitaj glavno polje A
9. Deklariraj pomoćno polje od P[101]
10. Postavi sve elemente pomoćnog polja na 0
11. Za i=0..N radi
12. P[A[i]]=P[A[i]]+1
13. k = 0
14. Za i=0..100 radi
15. Za j=1..P[i]
16. A[k] = i
17. k=k+1
18. Ispiši polja A

S obzirom na prethodno opisani problem izradili smo pseudokod i sada kada smo razradili logiku izvršavanja našeg algoritma možemo izraditi C++ kod.

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

int main () {

    int n;
    // unos broja elemenata
    do {
       cout << "N = ";
       cin >> n;
    } while (n<1 || n>100);

    // dinamicka alokacija polja
    int *A = new int [n];

    // unos vrijednosti elemenata u polje
    for (int i = 0; i<n; i++)
        do {
           cout << "A[" << i << "] = ";
           cin >> A[i];
        } while (A[i]<0 || A[i]>100);

    // deklaracija pomocnog polja
    int P[101];

    // inicijalizacija nul polja
    for (int i=0; i<101; i++) P[i] = 0;
    // pobrojavanje pojavnosti vrijednosti
    for (int i=0; i<n; i++) P[A[i]]++;

    int k = 0;
```

```
    // unos sortiranih elemenata u originalno polje
    for (int i=0; i<101; i++)
        for (int j=1; j<=P[i]; j++)
            A[k++] = i;

    // ispis elemenata polja
    for (int i=0; i<n; i++)
        cout << A[i] << " ";

    // dealokacija resursa
    delete [] A;

    cout << endl;
    system("pause");

    return 0;
}
```

Primijetite da smo u ovom primjeru polje alocirali dinamički, pa ako se može jednodimenzionalno polje alocirati dinamički postavlja se pitanje a što je sa dvodimenzionalnim? Naravno, isto je moguće i radi se na sljedeći način:

```
char (*a)[20] = new char [n][20];
```

Kod dinamičke alokacija dvodimenzionalnog polja imajte na umu da samo prva dimenzija može biti varijabilna.

Algoritam sortiranja koji smo koristili u prethodnom primjeru naziva se sortiranje prebrojavanjem. Nedostaci sortiranja prebrojavanjem su ti što ovo sortiranje nije efikasno ako elementi glavnog polja poprimaju vrijednosti iz velikog intervala cijelih brojeva te nije primjenjivo na decimalne brojeve.

Do sada smo vidjeli da pokazivači mogu pokazivati na razne strukture međutim između ostaloga pokazivači mogu pokazivati i ne druge pokazivače. Tako da recimo jedan pokazivač (p2) može sadržavati memorijsku adresu drugog pokazivača (p), tako da operatorom dereferencijacije dolazimo do sadržaja pokazivača (p) s time da je taj sadržaj isto memorijska adresa a dodavanjem još jednog operatora dereferencijacije dolazimo do vrijednosti koja se nalazi na memorijskoj lokaciji na koju pokazuje pokazivač (p).

Ilustrirajmo ovaj primjer na sljedećem C++ kodu.

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

int main()
{
    int i = 10;
    int *p = &i;
    int **p2 = &p;
```

```
    cout << p2 << " " << *p2 << " " << **p2 << " " <<
    **p2+10 << endl;

    system("pause");

    return 0;
}
```

Zadaci za vježbu

- Objasnite što su i čemu služe pokazivači.
- Čemu služi operator dereferencijacije?
- Koja je veza između polja i pokazivača?
- Izradite sljedeći algoritam uz pomoć pokazivača i dinamičkog alociranja memorijskog prostora:
  ULAZ: Unijeti prirodni broj *N* i *N* riječi.
  IZLAZ: Ispisati riječi od posljednje prema prvoj, a svaku riječ naopako.

Programski primjeri za laboratorijske vježbe

Zadatak 1.

Izradite program u kojem:

1. se računaju rezultati računskih operacija (+, -, ÷, ×, %, ^ te $\sqrt{\square}$). Argumente unosi korisnik. Isto tako potrebno je izraditi algoritam koji će pomnožiti svaki element unesenog polja određenim skalarom. Algoritme implementirati preko pokazivača te ispisati sve podatke na zaslon.

2. je potrebno u jednodimenzionalno polje, dinamički alocirano, unijeti određeni niz znakova. Korisnik određuje veličinu polja. Nakon unosa ispisuju se svi znakovi u polju na zaslon kao i broj njihova pojavljivanja te potom korisnik od svih ispisanih znakova odabire koji želi zamijeniti nakon čega se unosi novi znak s kojim će se zamijeniti prethodno odabrani. Rezultat zamjene ispisati na ekran. Algoritam je potrebno realizirati preko pokazivača.

3. se računa korijen svakog elementa matrice nakon korisnikova unosa iste. Broj redova i stupaca unosi korisnik (max veličina [100][100]). Nakon što korisnik unese elemente stupac po stupac i izračunaju se korijeni potrebno je obje dijagonale matrice sortirati silazno. Na kraju treba izračunati aritmetičku sredinu neparnih brojeva u obje dijagonale zajedno. Na zaslon je potrebno ispisati sve podatke uključujući i svaki uneseni stupac odmah nakon unošenja. Algoritam je potrebno realizirati preko pokazivača.

4. Korisnik unosi N prirodnih brojeva, N <= 40. Brojeve je potrebno sortirati uzlazno. Rješenje je potrebno izvesti pomoću vezane liste. Nakon sortiranje potrebno je provjeriti dali postoje brojevi koji se nalaze u rasponu ASCII kodova slova i ako da koji su to te koja slova su predstvaljena tim brojevima. Na zaslon ispisati sve potrebne podatke.

5. korisnik određuje veličinu jednodimenzionalnog polja. U alocirano polje je moguće unijeti i brojeve ali i znakove. Nakon unosa potrebno je parne brojeve sortirati uzlazno i postaviti ih na početak polja, neparne silazno i postaviti ih na kraj polja a znakove je potrebno sortirati abecedno (a - z) i smjestiti ih na sredinu polja. Isto tako između svake grupe elemenata je potrebno staviti delimiter ASCII koda 45. Algoritam realizirati preko pokazivača.