Programiranje s poljima u C++

Galović, Luka

Sveučilište u Rijeci – Odjel za informatiku

Jednopredmetna informatika

Luka Galović

Programiranje s poljima u C++

Završni rad

Mentor: prof.dr.sc. Maja Matetić

Rijeka, 2021.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 9. ožujak 2021.

Zadatak za završni rad

Pristupnik: Luka Galović

Naziv završnog rada: Programiranje s poljima u C++

Naziv završnog rada na eng. jeziku: Programming in C++ using arrays

Sadržaj zadatka:

Zadatak završnog rada je dati teorijski prikaz strukture podataka polja, operacija i algoritama koji se na njemu izvode. Potrebno je izraditi odabrane projektne zadatke, odnosno oblikovati programe u programskom jeziku C++ kao ilustraciju primjene postupaka izvedenih uporabom polja.

Mentor

Prof. dr. sc. Maja Matetić

Voditelj za završne radove

Doc. dr. sc. Miran Pobar

Zadatak preuzet: 9. ožujka 2021.

(potpis pristupnika)

Sažetak

U ovom radu, opisana su neka od svojstava jezika C++ s naglaskom na polja. Također su opisani neki algoritmi sortiranja i pretraživanja koji koriste jednodimenzionalna polja za implementaciju.

Praktični dio rada sastoji se od tri konzolne aplikacije. Jedna aplikacija koristi se za zbrajanje dvije matrice, druga aplikacija je implementacija Insertion sort algoritma dok je treća aplikacija implementacija binarnog pretraživanja polja.

Ključne riječi

C++, program, polje, indeksirana varijabla, algoritam, pretraživanje, sortiranje, programiranje, aplikacija, matrica, funkcija

Sadržaj

1.Uvod

Programski jezik C++ je jezik opće namjene koji radi na srednjoj razini što znači da možemo postići veću kontrolu nad hardverom nego npr. u programskom jeziku Java koji radi na višoj razini. Razvio ga je Danac Bjarne Stroustrup dok je radio u Bell Labs-u kao dodatak na programski jezik C kako bi se omogućio objektno orijentirani pristup te se zbog toga C++ još naziva i „C sa klasama“.

Napravljen je s idejom da se koristi za sistemsko programiranje i embedded sustave s naglaskom na brzinu, efikasnost i fleksibilnost.

Kroz godine je napravljeno puno dodatnih framework-ova i library-a te se danas u jeziku C++ može isprogramirati praktički bilo što, ali se zbog svoje kompleksnosti uglavnom koristi kod softvera gdje je bitna brzina poput sustava za upravljanje bazama podataka, web servera, računalnih igara i dr.

U ovom radu naglasak će biti na programiranju s poljima u jeziku C++. U prvom dijelu rada ću opisati sam jezik C++ s naglaskom na polja i njihovu primjenu kao i povijesni razvoj dok će drugi dio rada biti konzolna aplikacija za zbrajanje matrica, implementacija Insertion sort algoritma te implementacija binarnog pretraživanja.

2. Povijest razvoja programskih jezika

Prva računala koja su se pojavila bila su vrlo složena za korištenje te su ih koristili isključivo stručnjaci koji su bili osposobljeni za rad s računalom. Takav rad se sastojao od dva dijela: davanje uputa računalu i čitanje rezultata obrade. Dok čitanje rezultata nije bio toliki problem uvođenjem pisača na kojima su se ispisivali rezultati, unošenje uputa, tj. programiranje, se sastojalo od mukotrpnog unošenja nizova nula i jedinica. Takve programe bilo je isuviše složeno za pisati, čitati i ispravljati pa su se ubrzo pojavili prvi programski jezici nazvani „asembleri“ (engl. assemblers).

U takvim jezicima svaka strojna naredba predstavljena je nekom kraticom koja je razumljiva ljudima koji čitaju program. Neke od tih kratica su: ADD koja se koristiza zbrajanje, MOV koja se koristi za premještanje podatka s jednog na drugo mjesto u memoriji i dr. Tako se postigla bolja čitljivost programa ali i dalje je bilo složeno pisati i ispravljati programe jer su se računalu morale davati najdetaljnije upute za svaku operaciju. Taj problem kasnije će dovesti i do pojave programskog jezika C++ koji će osloboditi programera nekih rutinskih poslova te mu dopustiti da se usredotoči na problem koji rješava.

Godine 1972. dolazi do pojave jezika C, iz kojeg će se kasnije razviti C++. C je bio jezik opće namjene koji je svojim stupanjem na scenu postigao ogroman uspjeh. Neki od razloga za to su: C je bio jednostavan za učenje u usporedbi s asemblerima, omogućio je modularno pisanje programa te se programski kod vrlo brzo izvodio. Početkom 90-ih godina 20. stoljeća, 99%

komercijalnog softvera bilo je napisano u C-u, eventualno dopunjeno strojnim jezikom u nekim kritičnim dijelovima.

Razvoj tehnologije i softvera je napredovao pa su se stvari na tom području počele mijenjati. Projekti od nekoliko stotina tisuća i više linija koda na kojima rade timovi od desetak ili sto ljudi postali su učestalost pa su se počeli uvoditi dodatne stvari kojima bi se takvi projekti pojednostavili za izradu i održavanje, te kojima bi se omogućila iskoristivost koda u više različitih projekata.

Bjarne Stroustrup zaposlio se 1979. godine u Bell Labs (kasnije AT&T) te je započeo rad na jeziku „C s klasama“ (engl. C with classes) koji će kasnije postati C++. Pri izradi je uzeo dobra svojstva iz tadašnjih jezika: Simula, Clu, Algol68 i Ada, dok je za osnovu uzeo jezik C, koji je već bio u širokoj upotrebi, a isto tako je stvoren u Bell Labs.

3. Osnovna svojstva jezika C++

Kako je već spomenuto u uvodu, C++ je jezik opće namjene i srednje razine. Programski kod napisan u C++ se prevodi odnosno kompajlira (engl. compile) u strojni kod razumljiv računalu pomoću programa koji se naziva kompajler (engl. Compiler).

Najvažnija svojstva jezika C++ koja omogućuju objektno orijentirani pristup su:

1. Enkapsulacija (engl. Encapsulation)
2. Nasljeđivanje (engl. Inheritance)
3. Polimorfizam (engl. Polymorphism)

3.1. Enkapsulacija

Enkapsulacija se koristi kako bi sakrili, odnosno onemogućili direktan pristup varijablama definiranim u klasi drugim varijablama i funkcijama izvan klase te na taj način dobivamo viši nivo sigurnosti. Kako bi pristupili enkapsuliranim varijablama koristimo se metodama definiranim u istoj klasi poznatim kao SETTER i GETTER metode. U C++ se enkapsulacija radi tako da se varijable deklariraju ili definiraju u private dijelu čiji su članovi skriveni, tj. ne može im se pristupiti izvan klase u kojoj su definirani.

Slika 1 prikazuje primjer enkapsulacije. Varijabla „OIB“ tipa integer deklarirana je u private dijelu klase „Osoba“ te se njoj direktno može pristupiti samo unutar te klase. Kako bi pristupili toj varijabli izvan klase, definirali smo setter metodu „setOIB(c)“ kojom pridružujemo vrijednost te getter metodu „getOIB()“ kojom dohvaćamo vrijednosti u public dijelu klase.

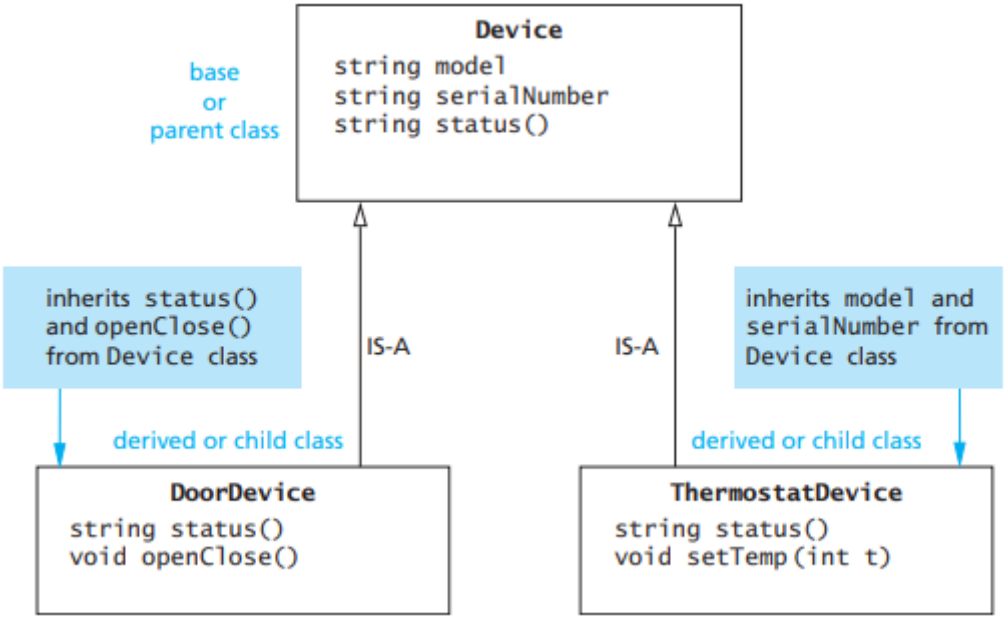
```cpp
#include <iostream>

using namespace std;

class Osoba {
private:
    int OIB;

public:
    // metoda za postavljanje varijabli
    void setOIB(int c){
        OIB = c;
    }

    // metoda za dohvaćanje varijabli
    int getOIB(){
        return OIB;
    }

};

int main()
{
    Osoba os;
    os.setOIB(1112223);
    int ob = os.getOIB();
    cout << ob;
    return 0;
}
```

```
1112223
Process returned 0 (0x0)   execution time : 0.020 s
Press any key to continue.
```

Slika 1. Enkapsulacija

3.2. Nasljeđivanje

Nasljeđivanje je jedna od najmoćnijih značajki jezika C++ kojom izvodimo jednu klasu iz druge. Nasljeđivanje je proces tokom kojeg je nova klasa, poznata kao izvedena klasa, napravljena iz druge klase koja se naziva bazna klasa. Izvedena klasa automatski ima sve varijable i funkcije koje su članovi bazne klase i može imati dodatne funkcije i/ili varijable. Također možemo reći da je bazna klasa roditelj, a izvedena klasa dijete. Kako bi ilustrirali korisnost nasljeđivanja, zamislimo da smo postavili kućni automatizacijski sustav kojim su vrata od garaže i termostat mrežno spojeni i kojima pristupamo sa računala. To bi puno lakše za koristiti kada bi imali konzistentno sučelje za te odvojene uređaje. Nadalje, svaki uređaj mora imati broj modela i serijski broj. Moguće je i da svaki uređaj ima način da se ispita njegov status. To bi se moglo modelirati klasom *Device* koja ima varijable za broj modela i serijski broj, te funkciju koja vraća status uređaja. Ideja je da ova klasa sadrži funkcije i svojstva koja imaju svi uređaji. Nadalje garažna vrata za razliku od termostata imaju funkciju za otvaranje i zatvaranje. To možemo oblikovati tako da napravimo izvedenu klasu *DoorDevice* iz klase *Device* koja će imati sve varijable i funkcije klase *Device* ali ćemo dodati još funkciju *openClose*(). Slično tako definirat ćemo klasu *ThermostatDevice* izvedenu iz klase *Device* za termostat.Termostat će također imati broj modela i serijski broj kao i funkciju za status ali ćemo još dodati i funkciju za postavljanje temperature. Također definiramo veze nasljeđivanja. Nakon što kreiramo objekt tipa *DoorDevice* ili objekt tipa *ThermostatDevice* imat ćemo pristup funkcijama i varijablama definiranim u klasi *Device*. Kako bi sve što smo opisali izgledalo na modelu prikazano je na Slici 2.



Slika 2. Model nasljeđivanja

Sljedeći primjer je nešto jednostavniji. Definirat ćemo baznu klasu *Vozilo* koja će sadržavati naziv marke i funkciju za trubljenje. Iz klase *Vozilo* ćemo napraviti izvedenu klasu *Auto* koja će sadržavati sve članove koje sadrži klasa *Vozilo* ali ćemo na to još dodati ime modela.

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

class Vozilo {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};

class Auto: public Vozilo {
  public:
    string model = "Mustang";
};

int main() {
  Auto mojAuto;
  mojAuto.honk();
  cout << mojAuto.brand + " " + mojAuto.model;
  return 0;
}
```

```
C:\Users\lukag\Desktop\ZavrÜni\ingh\bin\Debug\i...
Tuut, tuut!
Ford Mustang
Process returned 0 (0x0)   execution time : 0.007 s
Press any key to continue.
```

Slika 3. Nasljeđivanje

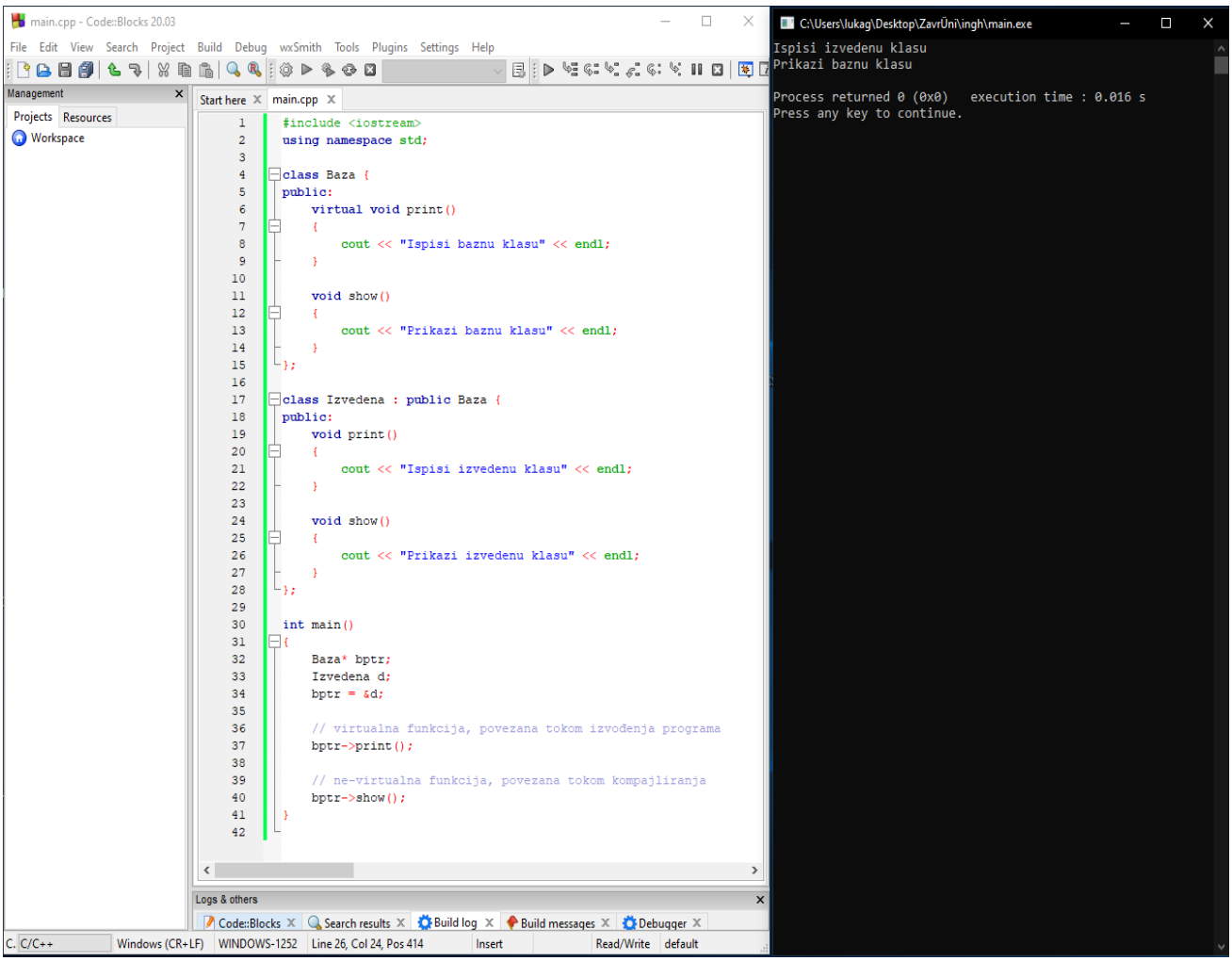
3.3. Polimorfizam

Polimorfizam se odnosi na sposobnost povezivanja više značenja s jednim funkcijskim imenom koje se izvodi mehanizmom koji se naziva *kasno povezivanje (*engl. late binding*)*. Ono omogućava definiranje operacija ovisnih o tipu.

3.3.1. Virtualna funkcija

Virtualna funkcija je funkcija koja je deklarirana unutar bazne klase i redefinirana, tj. preopterećena od izvedene klase. Kada se referiramo na objekt izvedene klase koristeći pokazivač ili reference na baznu klasu, možemo pozivati virtualnu funkciju za taj objekt i izvršiti verziju funkcije iz izvedene klase.

- Virtualna funkcija osigurava da se za objekt poziva korektna funkcija, bez obzira na tip reference (ili pokazivača) koji se koristi za funkcijski poziv
- Uglavnom se koriste kako bi se postigao polimorfizam
- Virtualna funkcija se deklarira ključnom riječi virtual unutar bazne funkcije
- Poziv funkcije vrši se u vrijeme izvođenja



Slika 4. Virtualna funkcija

4.Polja

Polja se koriste za obradu kolekcije podataka istog tipa, poput liste temperature, liste imena, liste cijelih brojeva i sl. U ovom poglavlju će biti predstavljeno više tehnika definiranja i rada s poljima u jeziku C++ kao i tehnike pri dizajniranju algoritama i programa koji koriste polja.

4.1. Uvod u polja

Zamislimo da trebamo rezultate nekog mjerenja koji sadrže 100 podataka obraditi te svaki podatak treba proći kroz iste računske operacije. Kada bi smo koristili zasebne varijable za svaki podatak morali bi definirati 100 različitih varijabli te 100 različitih blokova naredbi što nije ni malo praktično. Puno efikasnije je smjestiti sve podatke pod zajednički naziv, a pojedini podatak dohvaćati pomoću indeksa. Kako bi smo to ostvarili koristimo se poljima podataka (engl. arrays).
Polja možemo gledati kao listu varijabli istog tipa pod jednim imenom koje se može deklarirati u jednoj liniji koda.

4.2. Deklaracija i referenciranje polja

U jeziku C++, polje koje se sastoji od 10 varijabli tipa float može se deklarirati na sljedeći način:

```
int temperature[10];
```

Pojedine varijable u polju nazivaju se indeksirane varijable. Broj u uglatoj zagradi naziva se indeks. U većini programskih jezika, pa tako i u C++, indeksi se počinju brojati od 0. Broj indeksiranih varijabli u polju se naziva deklarirana veličina polja, ili jednostavnije veličina polja.
Moguće je deklarirati polje u istoj liniji sa varijablama:

```
int n, temperature[10], m;
```

Indeksiranu varijablu možemo koristiti na istim mjestima gdje možemo i običnu.
Na primjeru ispod varijabli t pridružujemo 6. element polja temperature. Uočavamo kako je indeks 6. elementa broj 5.

```
int t = temperature[5];
```

Također, indeks unutar uglatih zagrada ne mora biti strogo određeni broj. Možemo koristiti računske operacije:

```
int a = temperature[5+1];
```

Kod definicije polja, vrijednosti pišemo unutar vitičastih zagrada desno od znaka pridruživanja:

```
int temperature = {38, 25, 32, 33, 45, 31, 21, 19, 24, 14};
```

Isto tako možemo koristiti i već definirane varijable, što posebno koristi kada program ima interakciju sa korisnikom. Na Slici 5 prikazan je program koji od korisnika traži unos cijelog broja te potom ispisuje varijablu iz polja čiji je indeks jednak unesenom broju.

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



int main()
{
    int n;
    int temperature[10] = {38, 25, 32, 33, 45, 31, 21, 19, 24, 14}
    cout << "Unesi indeks: ";
    cin >> n;
    cout << temperature[n];
}
```

```
Unesi indeks: 3
33
Process returned 0 (0x0)   execution time : 22.933 s
Press any key to continue.
```

Slika 5. Interakcija sa korisnikom

Zamislimo da trebamo unijeti ili učitati vrijednosti iz polja s n elemenata. Kada bismo posebno unosili ili učitavali vrijednosti iz polja trebali bi n različitih blokova programa što nije ni malo praktično. Tu nastupa for petlja koja je idealna za manipulaciju poljima.
U primjeru na slici 6 pomoću for petlje unosimo n elemenata u polje te ih potom ispisujemo pri čemu smo varijabli n pridružili broj 5.

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



int main()
{
    int n =5;
    int temperature[n];
    for(int i=0; i<n; i++) {
        cout << "Unesi " << i+1 << ".element: ";
        cin >> temperature[i];
    }
    for(int j=0; j<n; j++) {
        cout << temperature[j] << " ";
    }
}
```

```
C:\Users\lukag\Desktop\ZavrUni\ingh\main.exe
Unesi 1.element: 2
Unesi 2.element: 6
Unesi 3.element: 77
Unesi 4.element: 32
Unesi 5.element: 1
2 6 77 32 1
Process returned 0 (0x0)   execution time : 3.355 s
Press any key to continue.
```

Slika 6. Iteracija kroz polje pomoću for petlje

4.3. Polja u memoriji

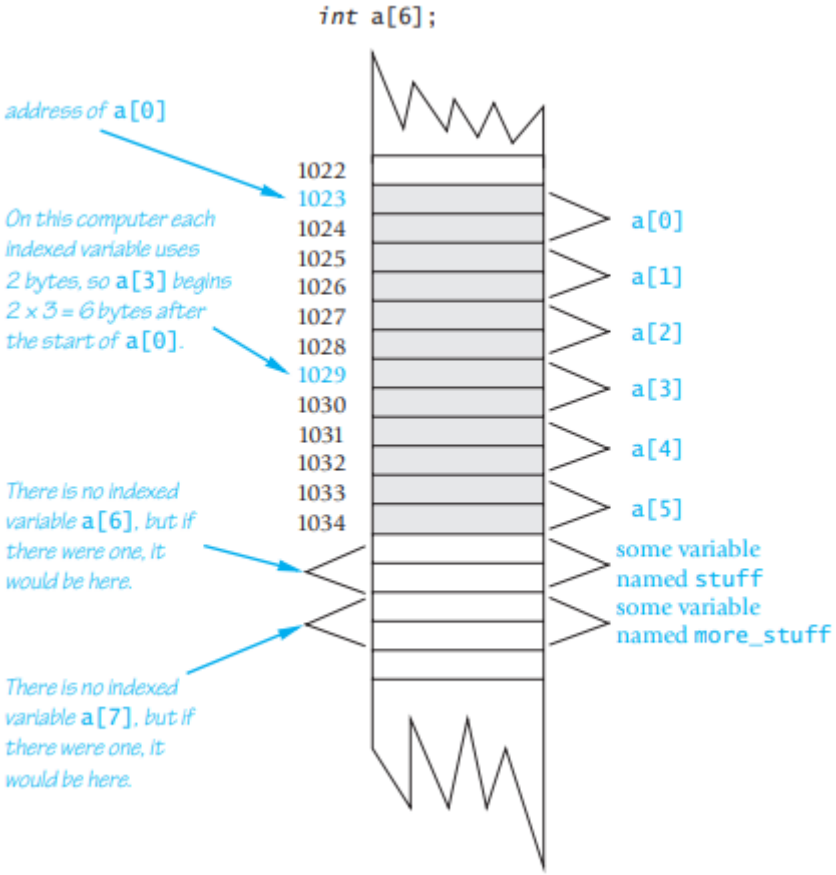
Prije nego objasnim kako se polja spremaju u memoriju, objasnit ću kako se jednostavna varijabla sprema u memoriju.
Računalna memorija sastoji se od niza numeriranih lokacija koje se nazivaju bajtovi. Broj bajta naziva se adresa u memoriji. Jednostavna varijabla zapisana je u memoriji kao broj uzastopnih bajtova. Broj bajtova određen je tipom podataka jer su tipovi različitih veličina.
Neki od tipova i njihove veličine u memoriji:

- int – 4 bajta
- float – 4 bajta
- double – 8 bajtova
- short int – 2 bajta
-

Kada deklariramo polje tipa double i veličine 5 prikazano na primjeru ispod, računalo u memoriji rezervira prostor za 5 varijabli tipa double, odnosno ukupno 40 bajtova.

```
int p[5];
```

Jedna od najčešćih grešaka kod rada s poljima je pokušaj dohvaćanja nepostojećeg indeksa. Uzmimo za primjer da samo deklarirali polje veličine 5. Ukoliko želimo dohvatiti varijablu polje[i], iterator mora biti 0, 1, 2, 3, 4 ili 5. Ukoliko je iterator nije jedan od navedenih indeksa doći će do greške. Tada kažemo da je takav izraz "ilegalan".
Kod većine sustava, rezultat ilegalnog indeksa polja dovest će do nečega pogrešnog pri čemu nećemo dobiti niti upozorenje.
Napadači su pronašli način da iskoriste ovu rupu kako bi provalili u software. 2011. godine je Common Weakness Enumeration (CWE)/SANS institute identificirao ovu vrstu greške kao treću najopasniju programersku grešku.



Slika 7. Polja u memoriji

Uzmimo za primjer kada imamo tipičan sustav, polje a je deklarirano kao na slici 7. ispod i program sadrži sljedeće:

a[i] = 238;

Pretpostavimo da vrijednost i, iz nekog razloga ispadne 7. Računalo nastavlja rad kao da je a[7] ispravna indeksirana varijabla. Računalo izračunava adresu gdje bi a[7] bio i stavlja vrijednost 238 na tu lokaciju u memoriji. Međutim, ne postoji indeksirana varijabla a[7] te memorija prima 238 kao da pripada nekoj drugoj varijabli, recimo varijabli *more_stuff*. Varijabla *more_stuff* je tada nenadano izmjenjena. Ova situacija ilustrirana je na slici 7.

4.4. Inicijalizacija polja

Polje može biti inicijalizirano kada je deklarirano. Kada inicijaliziramo polje, vrijednosti za pojedine indeksirane varijable stavljamo unutar vitičastih zagrada i odvajamo ih zarezima. Na primjer:

```
int djeca[3] = {4, 13, 2};
```

Ovakva deklaracija je identična sljedećem:

```
int djeca [3];
djeca [0] = 4;
djeca [1] = 13;
djeca [2] = 2;
```

Ukoliko unutar vitičastih zagrada napišemo manje vrijednosti nego je veličina deklariranog polja, prve indeksirane varijable, počevši od 0, će poprimiti te vrijednosti dok će ostale poprimiti vrijednost 0.
Ako inicijaliziramo polje pri deklaraciji, ne moramo pisati veličinu unutar uglatih zagrada, već će veličina polja automatski biti određena.

Na primjer:

```
int b[] = {1, 2, 3};
```

je isto kao da smo napisali

```
int b[3] = {1, 2, 3};
```

4.5. Polja u funkcijama

Pri prosljeđivanju argumenata funkciji možemo koristiti indeksiranu varijablu polja ili cijelo polje.

4.5.1. Prosljeđivanje indeksirane varijable kao funkcijskog argumenta

Indeksirana varijabla može biti funkcijski argument na isti način kako to može biti i bilo koja druga varijabla. Uzmimo za primjer da imamo program koji sadrži sljedeće deklaracije:

```
int i, n, a[10];
```

Ako *moja_funkcija* prima jedan argument tipa *int*, tada će sljedeće biti legalno:

```
moja_funkcija(n);
```

Kako je indeksirana varijabla polja a također varijabla tipa *int*, kao i *n*, sljedeće će biti legalno:

```
my_function(a[i]);
```

Ako je i=3, tada je argument funkcije a[3]. S druge strane, ako je i=0, onda će ovaj poziv biti ekvivalentan sljedećem:

```
my_function(a[0]);
```

Na slici 8. prikazan je primjer gdje se indeksirana varijabla koristi kao argument funkcije. Napisani program prikazuje 5 dodatnih dana odmora za svakog od 3 zaposlenika neke firme. Primjetimo da funkcija *adjust_days* ima formalni parametar pod nazivom *old_days* koji je tipa *int*. U main dijelu programa funkcija se poziva s argumentom vacation[number] za različite vrijednosti za number. U ovom primjeru indeksirane varijable su call-by-value argumenti. Iste opaske vrijede i za call-by-reference argumente. Indeksirane varijable mogu biti call-by-value argument ili call-by-reference argument.

```
1   //Illustrates the use of an indexed variable as an argument.
2   //Adds 5 to each employee's allowed number of vacation days.
3   #include <iostream>
4   const int NUMBER_OF_EMPLOYEES = 3;

5   int adjust_days(int old_days);
6   //Returns old_days plus 5.

7   int main( )
8   {
9       using namespace std;
10      int vacation[NUMBER_OF_EMPLOYEES], number;
11      cout << "Enter allowed vacation days for employees 1"
12           << " through " << NUMBER_OF_EMPLOYEES << ":\n";
13      for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
14          cin >> vacation[number - 1];
15      for (number = 0; number < NUMBER_OF_EMPLOYEES; number++)
16          vacation[number] = adjust_days(vacation[number]);
17      cout << "The revised number of vacation days are:\n";
18      for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
19          cout << "Employee number " << number
20               << " vacation days = " << vacation[number-1] << endl;
21      return 0;
22  }

23  int adjust_days(int old_days)
24  {
25      return (old_days + 5);
26  }
```

Slika 8. Indeksirana varijabla kao argument funkcije

Na slici 9. prikazan je rezultat gore napisanog programa.

```
Enter allowed vacation days for employees 1 through 3:
10 20 5
The revised number of vacation days are:
Employee number 1 vacation days = 15
Employee number 2 vacation days = 25
Employee number 3 vacation days = 10
```

Slika 9. Rezultat programa za indeksiranu varijablu kao argument funkcije

4.5.2. Prosljeđivanje cijelog polja kao funkcijskog argumenta

Funkcija može imati formalni parametar za cijelo polje te u tom slučaju dodajemo uglate zagrade iza imena varijable koja predstavlja polje kako bi računalo znalo da se radi o polju. Međutim, formalni parametar za cijelo polje nije niti call-by-value niti call-by-reference parametar, nego je potpuno nova vrsta parametra koji se naziva *parametar polja* (engl. array parameter).

Funkcija definirana na slici 10. ima jedan *array parameter "a",* koji će biti zamijenjen sa cijelim poljem kada se funkcija pozove.
Kada je funkcija *fill_up* pozvana ona mora imati dva argumenta. Prvi argument je polje dok je drugi argument deklarirana veličina polja.

Function Declaration

```
void fill_up(int a[], int size);
//Precondition: size is the declared size of the array a.
//The user will type in size integers.
//Postcondition: The array a is filled with size integers
//from the keyboard.
```

Function Definition

```
//Uses iostream:
void fill_up(int a[], int size)
{
    using namespace std;
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    size--;
    cout << "The last array index used is " << size << endl;
}
```

Slika 10. Funkcija sa parametrom polja

5. Programiranje s poljima

U ovom dijelu rada opisat ćemo upotrebu polja u algoritmima sortiranja i pretraživanja.

5.1. Djelomično popunjena polja

Potrebna veličina polja često nije poznata dok se program piše ili veličina varira od jednog pokretanja do drugog. Jednostavan način da riješimo ovu situaciju je da deklariramo polje maksimalne veličine koju će program u nekom trenutku trebati.

Djelomično popunjena polja zahtijevaju više pažnje. Program mora pratiti koliko je mjesta u polju zauzeto i ne smije referencirati indeksiranu varijablu koja nema definiranu vrijednost. Program na slici 11. ilustrira ovu situaciju. Program čita listu rezultata golf igre i prikazuje koliko se pojedini rezultat razlikuje od prosjeka. Program će raditi za liste od minimalno jednog, do maksimalno 10 rezultata. Rezultati se spremaju u polje *score*, koje ima 10 indeksiranih varijabli, ali program koristi onoliko koliko treba. Varijabla *number_used* čuva podatak koliko puno elemenata je spremljeno u polju. Elementi su spremljeni na pozicijama *score[0]* do *score[number_used -1]*.

```
1   //Shows the difference between each of a list of golf scores and their average.
2   #include <iostream>
3   const int MAX_NUMBER_SCORES = 10;
4   void fill_array(int a[], int size, int& number_used);
5   //Precondition: size is the declared size of the array a.
6   //Postcondition: number_used is the number of values stored in a.
7   //a[0] through a[number_used - 1] have been filled with
8   //nonnegative integers read from the keyboard.
9   double compute_average(const int a[], int number_used);
10  //Precondition: a[0] through a[number_used - 1] have values; number_used> 0.
11  //Returns the average of numbers a[0] through a[number_used - 1].
12  void show_difference(const int a[],int number_used);
13  //Precondition: The first number_used indexed variables of a have values.
14  //Postcondition: Gives screen output showing how much each of the first
15  //number_used elements of a differs from their average.
16  int main( )
17  {
18      using namespace std;
19      int score[MAX_NUMBER_SCORES], number_used;
20      cout << "This program reads golf scores and shows\n"
21           << "how much each differs from the average.\n";
22
23      cout << "Enter golf scores:\n";
24      fill_array(score, MAX_NUMBER_SCORES, number_used);
25      show_difference(score, number_used);
26      return 0;
27  }
28  //Uses iostream:
29  void fill_array(int a[], int size, int& number_used)
30  {
31      using namespace std;
32      cout << "Enter up to " << size << " nonnegative whole numbers.\n"
33           << "Mark the end of the list with a negative number.\n";
34      int next, index = 0;
35      cin >> next;
36      while ((next >= 0) && (index < size))
37      {
38          a[index] = next;
39          index++;
40          cin >> next;
41      }
42      number_used = index;
43  }
```

Slika 11. Program za golf rezultate

```
44    double compute_average(const int a[], int number_used)
45    {
46        double total = 0;
47        for (int index = 0; index < number_used; index++)
48            total = total + a[index];
49        if (number_used> 0)
50        {
51            return (total/number_used);
52        }
53        else
54        {
55            using namespace std;
56            cout << "ERROR: number of elements is 0 in compute_average.\n"
57                 << "compute_average returns 0.\n";
58            return 0;
59        }
60    }
61    void show_difference(const int a[], int number_used)
62    {
63        using namespace std;
64        double average = compute_average(a, number_used);
65        cout << "Average of the " << number_used
66             << " scores = " << average << endl
67             << "The scores are:\n";
68        for (int index = 0; index < number_used; index++)
69        cout << a[index] << " differs from average by "
70             << (a[index] - average) << endl;
71    }
```

Slika 11.1. Nastavak programa za golf rezultate

```
This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
69 74 68 -1

Average of the 3 scores = 70.3333
The scores are:
69 differs from average by -1.33333
74 differs from average by 3.66667
68 differs from average by -2.33333
```

Slika 11.2. Primjer dijaloga programa za golf rezultate

5.2. Pretraživanje polja

Uobičajeni programerski zadatak je pretražiti polje kako bi pronašli danu vrijednost. Na primjer, polje može sadržavati broj studenta za sve studente nekog tečaja. Kako bi odredili da li je određeni student uključen, polje se pretražuje kako bi vidjeli da li sadrži broj studenta. Program na slici 12. ispunjava polje te potom pretražuje polje za vrijednosti određene od strane korisnika. Ovo je primjer algoritma za sekvencijalno pretraživanje. Sekvencijalno pretraživanje je najizravniji oblik pretraživanja polja. Program pretražuje polje od prvog do zadnjeg element te provjerava da li pojedini element odgovara danoj vrijednosti.

U ovom primjeru funkcija *search* se koristi kako bi pretražili polje. Funkcija je dizajnirana da vraća indeks elementa ukoliko je vrijednost pronađena. U slučaju da nije, funkcija vraća vrijednost “-1”.

```
1   //Searches a partially filled array of nonnegative integers.
2   #include <iostream>
3   const int DECLARED_SIZE = 20;
4   void fill_array(int a[], int size, int& number_used);
5   //Precondition: size is the declared size of the array a.
6   //Postcondition: number_used is the number of values stored in a.
7   //a[0] through a[number_used - 1] have been filled with
8   //nonnegative integers read from the keyboard.
9   int search(const int a[], int number_used, int target);
10  //Precondition: number_used is <= the declared size of a.
11  //Also, a[0] through a[number_used - 1] have values.
12  //Returns the first index such that a[index] == target,
13  //provided there is such an index; otherwise, returns -1.
14  int main( )
15  {
16      using namespace std;
17      int arr[DECLARED_SIZE], list_size, target;
18      fill_array(arr, DECLARED_SIZE, list_size);
19      char ans;
20      int result;
21      do
22      {
23          cout << "Enter a number to search for: ";
24          cin >> target;
25          result = search(arr, list_size, target);
26          if (result == -1)
27              cout << target << " is not on the list.\n";
28          else
29              cout << target << " is stored in array position "
30              << result << endl
31              << "(Remember: The first position is 0.)\n";
32          cout << "Search again?(y/n followed by Return): ";
33          cin >> ans;
34      } while ((ans != 'n') && (ans != 'N'));
35      cout << "End of program.\n";
36      return 0;
37  }
38  //Uses iostream:
39  void fill_array(int a[], int size, int& number_used)
40
```

Slika 12. Algoritam za sekvencijalno pretraživanje

```
41  int search(const int a[], int number_used, int target)
42  {
43
44      int index = 0;
45      bool found = false;
46      while ((!found) && (index < number_used))
47          if (target == a[index])
48              found = true;
49          else
50              index++;
51
52      if (found)
53          return index;
54      else
55          return -1;
56  }
```

Slika 12.1. Nastavak algoritma za sekvencijalno pretraživanje

```
Enter up to 20 nonnegative whole numbers.
Mark the end of the list with a negative number.
10 20 30 40 50 60 70 80 -1
Enter a number to search for: 10
10 is stored in array position 0.
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 40
40 is stored in array position 3.
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 42
42 is not on the list.
Search again?(y/n followed by Return): n
End of program.
```

Slika 12.2. Primjer dijaloga programa koji koristi algoritam za sekvencijalno pretraživanje

5.3. Sortiranje polja

Jedan od najčešće susretanih problema kod programiranja je kako sortirati dano polje, poput liste prodaje koja treba biti sortirana od najmanjeg do najvećeg elementa ili od najvećeg do najmanjeg elementa ili kako sortirati listu riječi prema abecedi. U ovom poglavlju opisat ću funkciju *sort* koja sortira djelomično popunjeno polje brojeva od najmanjeg prema najvećem.

Deklaracija funkcije:

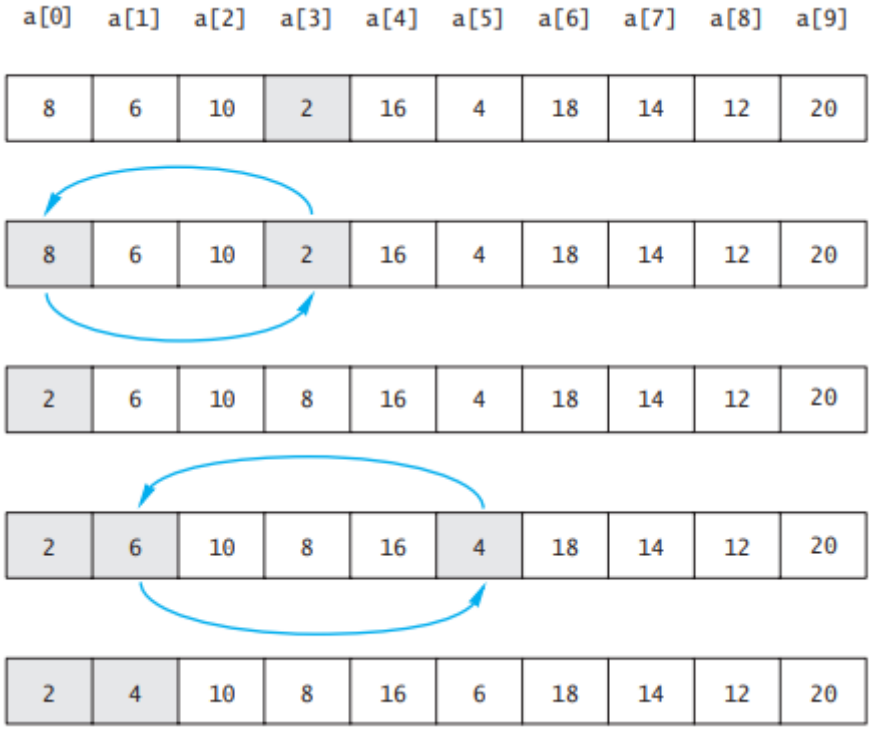
```
void sort(int a[], int number_used)
```

Varijablu *number_used* koristimo kako bi deklarirali veličinu polja *a*.
Funkcija *sort* preuređuje elemente polja tako da nakon svakog poziva funkcije su elementi poredani na sljedeći način:

```
a[0] ≤ a[1] ≤ a[2] ≤ ... ≤ a[number_used - 1]
```

5.3.1. Selection sort

Algoritam koji ću koristiti naziva se *Selection sort.* To je jedan od najjednostavnijih algoritama za sortiranje.



Slika 13. Ilustracija algoritma Selection sort

```
//Tests the procedure sort.
#include <iostream>

void fill_array(int a[], int size, int&number_used);
//Precondition: size is the declared size of the array a.
//Postcondition: number_used is the number of values stored in a.
//a[0] through a[number_used - 1] have been filled with
//nonnegative integers read from the keyboard.

void sort(int a[], int number_used);
//Precondition: number_used <= declared size of the array a.
//The array elements a[0] through a[number_used - 1] have values.
//Postcondition: The values of a[0] through a[number_used - 1] have
//been rearranged so that a[0] <= a[1] <= ... <= a[number_used - 1].

void swap_values(int &v1, int &v2);
//Interchanges the values of v1 and v2.

int index_of_smallest(const int a[], int start_index, int number_used);
//Precondition: 0 <= start_index < number_used. Referenced array elements have
//values.
//Returns the index i such that a[i] is the smallest of the values
//a[start_index], a[start_index + 1], ..., a[number_used - 1].

int main( )
{
    using namespace std;
    cout << "This program sorts numbers from lowest to highest.\n";

    int sample_array[10], number_used;
    fill_array(sample_array, 10, number_used);
    sort(sample_array, number_used);

    cout << "In sorted order the numbers are:\n";
    for (int index = 0; index < number_used; index++)
    cout << sample_array[index] << " ";
    cout << endl;

    return 0;
}

//Uses iostream:
void fill_array(int a[], int size, int&number_used)
```

<The rest of the definition of fill_array is given in Display 7.9.>

```
void sort(int a[], int number_used)
{
    int index_of_next_smallest;
    for (int index = 0; index < number_used - 1; index++)
```

Slika 14. Programska realizacija algoritma Selection sort

```
39        {//Place the correct value in a[index]:
40            index_of_next_smallest =
41                           index_of_smallest(a, index, number_used);
42            swap_values(a[index], a[index_of_next_smallest]);
43            //a[0] <= a[1] <=...<= a[index] are the smallest of the original array
44            //elements. The rest of the elements are in the remaining positions.
45        }
46    }
47
48    void swap_values(int& v1, int& v2)
49    {
50        int temp;
51        temp = v1;
52        v1 = v2;
53        v2 = temp;
54    }
55
56    int index_of_smallest(const int a[], int start_index, int number_used)
57    {
58        int min = a[start_index],
59            index_of_min = start_index;
60        for (int index = start_index + 1; index < number_used; index++)
61            if (a[index] < min)
62            {
63                min = a[index];
64                index_of_min = index;
65                //min is the smallest of a[start_index] through a[index]
66            }
67
68        return index_of_min;
69    }
```

Slika 14.1. Nastavak programske realizacije algoritma Selection sort

```
This program sorts numbers from lowest to highest.
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
80 30 50 70 60 90 20 30 40 -1
In sorted order the numbers are:
20 30 30 40 50 60 70 80 90
```

Slika 14.2. Primjer dijaloga Selection sort-a

5.3.2. Bubble sort

Bubble sort je jednostavan i općenit algoritam za sortiranje koji je sličan selection sortu. Ukoliko koristimo bubble sort za sortiranje polja u rastućem redosljedu, tada najveća vrijednost je poput mjehurića odnesena na kraj polja. Zbog toga se ovaj algoritam naziva Bubble sort.

Ako imamo nesortirano polje koje se sastoji od sljedećih elemenata:

```
{3, 10, 9, 2, 5}
```

Tada, nakon prvog prolaza, najveća vrijednost koja je u ovom slučaju 10, biti će prebačena na kraj polja.

Nakon prvog prolaza:

```
{3, 9, 2, 5, 10}
```

Nakon drugog prolaza:

```
{3, 2, 5, 9, 10}
```

Nakon trećeg prolaza:

```
{2, 3, 5, 9, 10}
```

Nakon četvrtog prolaza:

```
{2, 3, 5, 9, 10}
```

U ovom trenutku algoritam je završio. Preostali broj na početku polja ne treba biti pregledan jer je jedini preostao te je samim time najmanji.

Na primjeru ispod prikazana je programska realizacija algoritma *Bubble sort.*

```
//DISPLAY 7.13 Bubble Sort Program
//Sorts an array of integers using Bubble Sort.
#include <iostream>

void bubblesort(int arr[], int length);
//Precondition: length <= declared size of the array arr.
//The array elements arr[0] through a[length - 1] have values.
//Postcondition: The values of arr[0] through arr[length - 1] have
//been rearranged so that arr[0] <= a[1] <=  <= arr[length - 1].

int main()
{
    using namespace std;
    int a[] = {3, 10, 9, 2, 5, 1};

    bubblesort(a, 6);
    for (int i=0; i<6; i++)
    {
         cout << a[i] << " ";
    }
    cout << endl;
    return 0;
}

void bubblesort(int arr[], int length)
{
        // Bubble largest number toward the right
        for (int i = length-1; i > 0; i--)
                for (int j = 0; j < i; j++)
                        if (arr[j] > arr[j+1])
                        {
                                // Swap the numbers
                                int temp = arr[j+1];
                                arr[j+1] = arr[j];
                                arr[j] = temp;
                        }
}
```

Slika 15. Programska realizacija algoritma Bubble sort

```
1 2 3 5 9 10
```

Slika 15.1. Primjer dijaloga Bubble sort-a

6. Multidimenzionalna polja

Jezik C++ dopušta deklaraciju polja s više indeksa, odnosno više dimenzija.
Ponekad nam je potrebno imati polje s više od jednog indeksa. Uzmimo za primjer da želimo imati polje s 2 dimenzije.
Deklaracija polja tipa *char* s 30 redaka i 100 stupaca izgledalo bi ovako:

```
char stranica[30][100];
```

Indeksirane varijable za ovo polje imaju po 2 indeksa. Jedna od njih bi bila stranica[0,0], stranica[15][32] i dr. Indeks svake dimenzije treba biti zatvoren u svojoj uglatoj zagradi.

7. Primjena polja u praksi

U završnom dijelu ovoga rada izradio sam konzolnu aplikaciju za rad s matricama kako bih pokazao primjenu polja. Aplikacija se koristi za zbrajanje dviju matrica unesenih od strane korisnika.

7.1. Aplikacija za zbrajanje dviju matrica

Na početku programa deklarirali smo varijable *r* i *c* koje nam služe za spremanje veličine redaka i stupaca te smo deklarirali 2 dvodimenzionalna polja maksimalne veličine [100][100] iako mogu biti i manja, ovisno o korisniku.
Od korisnika se traži da unese broj redaka i stupaca matrica te se potom koriste dvije ugnježdene *for* petlje za unos elemenata prve, a potom i druge matrice.
Nakon što su elementi uneseni, pokrećemo ugnježdenu *for* petlju koja vrši zbrajanje matrica te zbroj elemenata spremamo u višedimenzionalno polje *sum*.
Na kraju pomoću još jedne ugnježdene *for* petlje ispisujemo polje *sum*, odnosno rezultat zbroja dviju matrica.

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

int main()
{
    int r, c, a[100][100], b[100][100], sum[100][100], i, j;

    cout << "Unesi broj redaka (izmedu 1 i 100): ";
    cin >> r;

    cout << "Unesi broj stupaca (izmedu 1 i 100): ";
    cin >> c;

    cout << endl << "Unesi elemente 1. matrice: " << endl;

    // Spremanje elemenata prve matrice koje korisnik unosi
    for(i = 0; i < r; ++i)
       for(j = 0; j < c; ++j)
       {
           cout << "Unesi element a" << i + 1 << j + 1 << " : ";
           cin >> a[i][j];
       }

    // Spremanje elemenata druge matrice koje korisnik unosi
    cout << endl << "Unesi elemente 2. matrice: " << endl;
    for(i = 0; i < r; ++i)
       for(j = 0; j < c; ++j)
       {
           cout << "Unesi element b" << i + 1 << j + 1 << " : ";
           cin >> b[i][j];
       }

    // Zbrajanje matrica
    for(i = 0; i < r; ++i)
        for(j = 0; j < c; ++j)
            sum[i][j] = a[i][j] + b[i][j];

    // Ispis rezultata zbroja matrica.
    cout << endl << "Zbroj 1. i 2. matrice je: " << endl;
```

Slika 17.1. Aplikacija za zbrajanje matrica

```
    for(i = 0; i < r; ++i)
        for(j = 0; j < c; ++j)
        {
            cout << sum[i][j] << "   ";
            if(j == c - 1)
                cout << endl;
        }

    return 0;
}
```

Slika 17.2. Aplikacija za zbrajanje matrica (str.2.)

```
Unesi broj redaka (izmedu 1 i 100): 4
Unesi broj stupaca (izmedu 1 i 100): 2

Unesi elemente 1. matrice:
Unesi element a11 : 1
Unesi element a12 : 5
Unesi element a21 : 22
Unesi element a22 : 3
Unesi element a31 : 77
Unesi element a32 : 23
Unesi element a41 : 1
Unesi element a42 : 23

Unesi elemente 2. matrice:
Unesi element b11 : 44
Unesi element b12 : 1
Unesi element b21 : 22
Unesi element b22 : 5
Unesi element b31 : 6
Unesi element b32 : 98
Unesi element b41 : 0
Unesi element b42 : 44

Zbroj 1. i 2. matrice je:
45  6
44  8
83  121
1  67
```

Slika 17.3. Primjer dijaloga aplikacija za zbrajanje matrica

7.2. Implementacija Insertion sort algoritma

Insertion sort je jednostavan algoritam sortiranja koji se može ilustrirati kao način kad kartamo pa raspoređujemo karte u ruci. Polje se dijeli na sortirani i nesortirani dio. Vrijednosti iz nesortiranog dijela se uzimaju i stavljaju na ispravnu poziciju u sortirani dio.
U programu ispod, ilustrirana je implementacija Insertion sort algoritma kroz funkciju *insertionSort*. Funkciji prosljeđujemo polje te varijablu *n* koja sadrži veličinu polja.

```
#include <iostream>
using namespace std;
void insertionSort(int arr[], int n)
{
    int i, temp, j;
    for (i = 1; i < n; i++)
    {
        temp = arr[i];
        j = i - 1;
        /* Pomakni elemente od arr[0 do i-1], koji su
        veci od trenutnog(temp), za jednu poziciju ispred
        trenutne pozicije
        */
        while (j >= 0 && arr[j] > temp)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = temp;
    }
}
// Pomocna funkcija koja nam sluzi za ispis polja velicine n
void ispis(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}
int main()
{
    int arr[] = { 5, 90, 15, 7, 6, 11, 25, 34 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Nesortirano polje: ";
    ispis(arr, n);
    insertionSort(arr, n);
    cout << "Sortirano polje: ";
    ispis(arr, n);
    return 0;
```

Slika 18.1. Implementacija Insertion sort algoritma

```
Nesortirano polje: 5 90 15 7 6 11 25 34
Sortirano polje: 5 6 7 11 15 25 34 90

Process returned 0 (0x0)   execution time : 0.007 s
Press any key to continue.
```

Slika 18.2. Primjer dijaloga za Insertion sort

7.3. Binarno pretraživanje

Binarno pretraživanje je algoritam pretraživanja koji se primjenjuje na sortirana polja. U primjeru ispod ilustriran je navedeni algoritam. Funkciji *binarySearch* prosljeđujemo polje, varijable *l* i *r* koje predstavljaju lijevu i desnu granicu te varijablu *x* koja predstavlja traženi element. Prvo uspoređujemo traženi element sa elementom u sredini polja. Ukoliko je traženi element u sredini polja, vraćamo indeks tog elementa. Inače, ako je traženi element veći od srednjeg, tada on može biti jedino u desnoj polovici polja pa lijevu ignoriramo. Analogno, ako je manji od srednjeg elementa, ignoriramo desnu polovicu polja.

```
#include <iostream>

using namespace std;
// Funkcija za binarno pretraživanje koja vraca lokaciju od x
// u proslijedjenom polju ukoliko postoji.
// Ukoliko x ne postoji program vraca -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int mid = l + (r - l) / 2;
        // Provjeravamo da li je x u sredini
        if (arr[mid] == x)
            return mid;
        // Ako je x veci, ignoriramo lijevu polovicu polja
        if (arr[mid] < x)
            l = mid + 1;
        // Ako je x manji, ignoriramo desnu polovicu polja
        else
            r = mid - 1;
    }

    // Ukoliko element nije pronaden vracamo -1
    return -1;
}
int main(void)
{
    int arr[] = { 8, 11, 54, 110, 115, 199 };
    int x = 199;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    if(result == -1) {
        cout << "Element se ne nalazi u polju";
    }
    else {
        cout << "Element se nalazi na mjestu: " << result;
    }

    return 0;
}
```

Slika 19.1. Implementacija algoritma za binarno pretraživanje

```
Element se nalazi na mjestu: 5
Process returned 0 (0x0)   execution time : 0.000 s
Press any key to continue.
```

Slika 19.2. Primjer dijaloga za binarno pretraživanje

8. Zaključak

Kroz ovaj rad obrađena su neka od osnovnih svojstava jezika C++ s naglaskom na polja i njihovu primjenu, kao i kratak opis razvoja programskih jezika te kako se došlo do jezika C++. U završnom dijelu ovog rada, ilustrirana je primjena polja kod zbrajanja matrica, sortiranja algoritmom Insertion sort te binarno pretraživanje. Jezik C++ ima vrlo široku primjenu, iako se danas većinom koristi kod aplikacija gdje je bitna brzina i efikasnost. Kako je C++ jezik srednje razine, ima vrlo široku primjenu u školama i na sveučilištima kako bi se prikazala implementacija algoritama, te kako neke stvari rade "ispod haube" poput pokazivača koje ne možemo vidjeti u jezicima više razine.

Literatura

- Savitch, W.J., Mock, K., Msanjila, S. and Muiche, L., 2015. Problem Solving with C++. Pearson.
- Julijan Šribar, Boris Motik, Demistificirani C++.