

STRUKTURE PODATAKA I ALGORITMI

Međuispit 1

CLASS

Klasa (eng. class) je korisnički – definiran tip podataka koji koristimo u našem programu. Klasa služi kao svojevrsni “blueprint” za kreiranje objekata. Članovi koji se nalaze u klasi su po default-u privatni, dok su u slučaju strukture javni.

Koja je razlika između klase i strukture ako su oboje “ista” stvar? **Nema je...** Razlika je u tome što točno planiramo raditi.

- Ako imamo potrebu za tipom čija je glavna namjena čuvanje podataka bez puno operacija na njima –
--> **Struktura**
- Za sve ostalo --> **Klasa**

Kako definiramo klasu? Klasa se definira na sljedeći način:

```
#pragma once
class Rectangle
{
private:
    double a, b;
public:
    // konstruktori
    Rectangle(double a, double b);

    // seteri
    void set_a(double a);
    void set_b(double b);

    // geteri
    double get_a();
    double get_b();

    // funkcije s kojima radimo
    double opseg();
    double površina();
};
```

Na slici se može vidjeti da se klasa radi po istom principu kao i struktura, jedina je razlike u tome što sada postoje **private** i **public** koji definiraju da li će se varijable moći koristiti samo unutar klase ili van nje.

- Varijable koje se nalaze u private dijelu mogu se koristiti samo unutar klase, ne i u mainu.
- Varijable koje se nalaze u public dijelu mogu se koristiti van klase, tj. u mainu.

Što sve spremamo u public dio klase, a što u private dio?

- **PRIVATE** – private sadržava sve vaše varijable koje ćete koristiti, bilo to int, double, float. Uz to, tu se stavljaju sve varijable koje se ne koriste u mainu.
- **PUBLIC** – u public dio se stavljaju sve metode (funkcije), konstruktori i destruktori koje definiramo i koje želimo koristiti u mainu

Kako kreiramo klasu?

Desni klik na projekt ---> Add ---> Class ---> Klasi damo neki naziv ---> OK ---> Dobili smo Header file (.h) i cpp file. (CPP file će kasnije biti objašnjen)

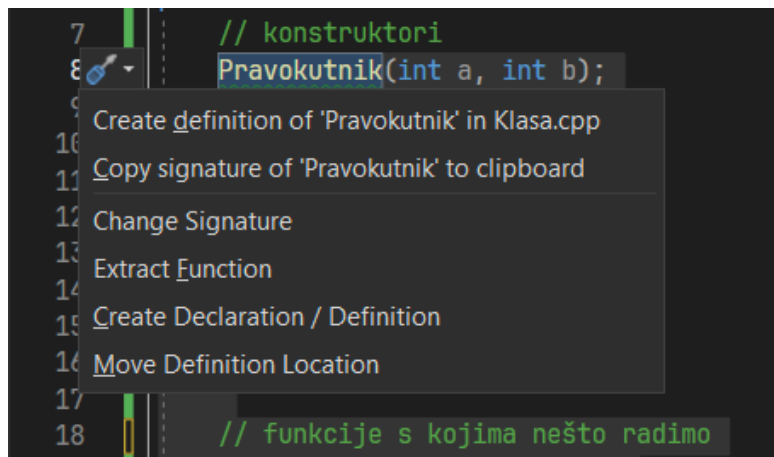
#pragma once dobijate po default-u u vašem header file-u, on služi kao include zaštita i njega ostavljate takvim kakav jest (detaljnije od ovoga nije potrebno znati).

Kako jednostavno deklariramo sve funkcije i inicijaliziramo varijable i metode?

1. KORAK – selectamo public dio

```
6 public:
7     // konstruktori
8     Pravokutnik(double a, double b);
9
10    // seteri
11    void set_a(double a);
12    void set_b(double b);
13
14    // geteri
15    double get_a();
16    double get_b();
17
18    // funkcije s kojima nešto radimo
19    double opseg(double a, double b);
20    double površina(double a, double b);
21 };
```

2. KORAK – kliknemo ALT + ENTER



3. KORAK – Kliknemo na Create Declaration / Definition

Kada smo to kliknuli, napokon ulazimo u nas .cpp file koji smo dobili uz .h file.

.cpp file bi vam inicijalno trebao izgledati ovako:

```
#include "Rectangle.h"

Rectangle::Rectangle(double a, double b)
{
}

void Rectangle::set_a(double a)
{
}

void Rectangle::set_b(double b)
{
}

double Rectangle::get_a()
{
    return 0.0;
}

double Rectangle::get_b()
{
    return 0.0;
}

double Rectangle::opseg()
{
    return 0.0;
}

double Rectangle::povrsina()
{
    return 0.0;
}
```

Na slici se može vidjeti da nema onih varijabli koje su definirane u .h file-u. Iako ih ne vidimo, one su tu i aktivno ih možemo koristiti u daljnjem radu.

Za inicijalizaciju varijabli postoji nekoliko metoda, dvije najvažnije su:

- Inicijalizacija konstruktorom
- Inicijalizacija seterima

Inicijalizacija seterima izgleda ovako:

```
void Rectangle::set_a(double a)
{
    this->a = a;
}

void Rectangle::set_b(double b)
{
    this->b = b;
}
```

Kao što se može primijetiti, za inicijalizaciju varijable a i b koristili smo pointer **this**. **This** je pointer na samoga sebe (u prijevodu pokazuje na našu varijablu). This pointer koristimo umjesto drugih načina iz razloga što ne stvaramo dodatne varijable (ili kopije varijabli) kako bi inicijalizirali varijablu.

Inicijalizacija konstruktorom izgleda ovako:

```
Rectangle::Rectangle(double a, double b)
{
    this->a = a;
    this->b = b;
}
```

Kasnije ćemo detaljnije objasniti kako konstruktori funkcioniraju, te kako zapravo ovo radi. Za sada se držimo inicijalizacije seterima.

Za dohvaćanje potrebnih varijabli koristimo getere koje smo definirali ranije. Getere nije preporučljivo previše koristiti jer je cilj napraviti program koji se neće u potpunosti "otvoriti" korisniku (u prijevodu, getane varijable postaju javne, a to nije nužno dobro).

Dohvaćanje geterima izgleda ovako:

```
double Rectangle::get_a()
{
    return a;
}

double Rectangle::get_b()
{
    return b;
}
```

Za što nam služe geteri? Suština getera je da vrati traženu varijablu. U ovom slučaju, ako u mainu pozovemo funkciju iz klase naziva **get_b()**, ona će ga dohvatiti i vratiti varijablu nama (kasnije ćemo demonstrirati kako to izgleda).

Defaultni konstruktor vs. Korisnički definiran konstruktor:

Konstruktor je po definiciji metoda koja stvara objekt na klasi. Konstruktor može biti definiran na više načina, a jedan od njih je onaj defaultni.

Defaultni konstruktor je automatski generiran konstruktor koji se poziva onog trenutka kada u main stavimo **Pravokutnik p1**; Defaultni konstruktor je izuzetno opasan. Zašto je opasan? Opasan je iz razloga što one varijable a i b koje smo prije definirali neće biti inicijalizirane, nego će se u njima naći smeće.

Korisnički definiran konstruktor je onaj konstruktor koji je jasno definiran u klasi i na kraju pozvan u mainu. To je u suštini konstruktor kojemu najčešće šaljemo vrijednosti pri pozivu. Korisnički definiranom konstruktoru kojemu nije poslana niti jedna varijabla možemo namjestiti da iste defaultno inicijalizira. Objasnimo pomoću slika...

```
Rectangle::Rectangle(double a, double b)
{
    this->a = a;
    this->b = b;
}
```

Ovo je korisnički definiran konstruktor. Kao što se može vidjeti, konstruktor prima nekakve dvije varijable, te ih inicijalizira na zadanu vrijednost. U ovom slučaju, kakvu god vrijednost pošaljemo konstruktoru za varijable a i b, on će ih inicijalizirati, te će njihova vrijednost biti ona koju smo definirali (ovo je ujedno i razlog zašto je ovo bolje od defaultnog).

Kako pozivamo korisnički definiran konstruktor?

Korisnički definiran konstruktor pozivamo kroz main na sljedeći način:

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

using namespace std;

int main()
{
    Rectangle r(1, 2);

    return 0;
}
```

Vidi li se sada povezanost između konstruktora u mainu i metode u .cpp file-u? Odgovor je naravno! Pozivom konstruktora **Pravokutnik p(1, 2);** mi smo zapravo poslali vrijednosti varijable a i varijable b.

Kako to dokazati?

Jednostavno! Princip je isti kao i kod struktura iz prvog semestra:

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

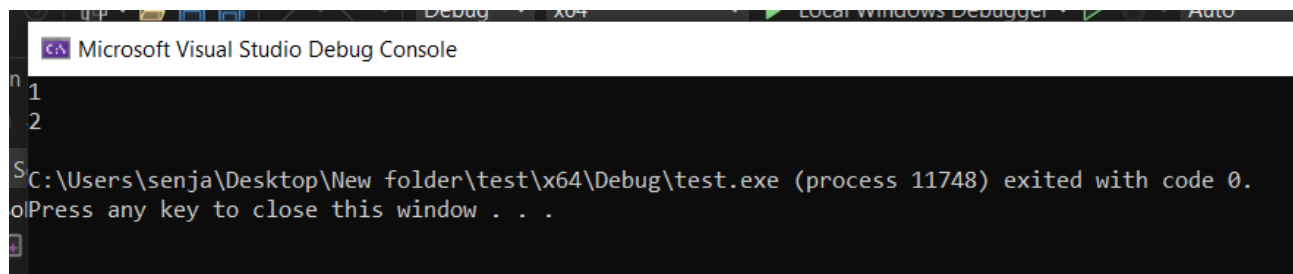
using namespace std;

int main()
{
    Rectangle p(1, 2);

    cout << p.get_a() << endl;
    cout << p.get_b() << endl;

    return 0;
}
```

Dokaz da konstruktor funkcioniра:

The image shows a screenshot of the Microsoft Visual Studio Debug Console. The window title is "Microsoft Visual Studio Debug Console". The console output shows the program's execution path: "C:\Users\senja\Desktop\New folder\test\x64\Debug\test.exe (process 11748) exited with code 0." followed by "Press any key to close this window . . .". The background is dark, and the text is light gray.

NAPOMENA:

Primjetite da u mainu stoji nova vrsta include-a. **#include "Pravokutnik.h"** nam omogućuje korištenje našeg header file-a i onoga što smo definirali u .cpp u mainu (zato i možemo pozvati **Pravokutnik p(1, 2);**). Header se uvijek uključuje u main ako ga želimo koristiti.

STRINGSTREAM

Uz sve što smo do sada prošli, postoji još jedan način kojim možemo ispisati ono što nas zanima, a to je stringstream. Stringstream je objekt, što u suštini znači da mu možemo dati neki naš naziv. Stringstream kao takav nije vezan za konzolu jer vraća string, pa je tako zapravo može pozvati bilo koji klijent, a ne samo konzola.

Stringstream još možemo opisati kao "bačvu" u koju stavljamo sve što mi želimo, te on na kraju sve to izbacuje kao jedan string

Da bi uopće koristili stringstream, moramo prvo napraviti metodu:

```
string to_string();
```

Da bi ova metoda radila moramo u naš header includeati **<string>** i ispod napisati **using namespace std;**

Zašto? Stringstream je stream stringova...

Kada smo to odradili, u našem .cpp file-u moramo uvesti novu vrstu includeati, a to je **<sstream>**. **<sstream>** omogućava korištenje stringstream-a, a princip rada sa stringstream-om je apsolutno isti kao i kod **cout-a**.

```
string Rectangle::to_string()
{
    stringstream ss;

    ss << get_a() << ", " << get_b();

    return ss.str();
}
```

MJERENJE VREMENA TRAJANJA ALGORITAMA

Kako bi smo mjerili vrijeme trajanja izvršenja nekog programa/algoritma koji smo napisali, koristimo za to ugrađene funkcije koje dolaze s novom vrstom headera **#include <chrono>**.

Chrono je library za mjerenje / korištenje vremena. Mi ćemo ga koristiti najviše za mjerenje trajanja nekog algoritma.

Kako se to radi?

```
#include <iostream>
#include <chrono>
#include "Rectangle.h"

using namespace std;
using namespace std::chrono;

int main()
{
    auto begin = high_resolution_clock::now();
    Rectangle p(1, 2);

    cout << p.get_a() << endl;
    cout << p.get_b() << endl;

    auto end = high_resolution_clock::now();

    cout << duration_cast <nanoseconds> (end - begin).count() << endl;

    return 0;
}
```

Na slici možemo vidjeti da smo prvo includali **<chrono>** koji nam omogućava korištenje objekata **high_resolution_clock::now()**, te funkcije **duration_cast <nanoseconds>**. Kada smo njega includali, možemo napisati dio koda koji će mjeriti vrijeme. Na slici su to **auto begin** i **auto end** koji naznačavaju početak i kraj mjerenja vremena u našem programu.

Sve što se nalazi između begina i enda upada u mjerenje.

Preko funkcije **duration_cast <nanoseconds> (end – begin).count()** ispisujemo izmjereno vrijeme.

Valja napomenuti da se ovaj kod iznad može znatno skratiti eliminacijom **chrono::** dijelova u kodu, a to se radi tako da u naš kod upišemo **using namespace std::chrono**.

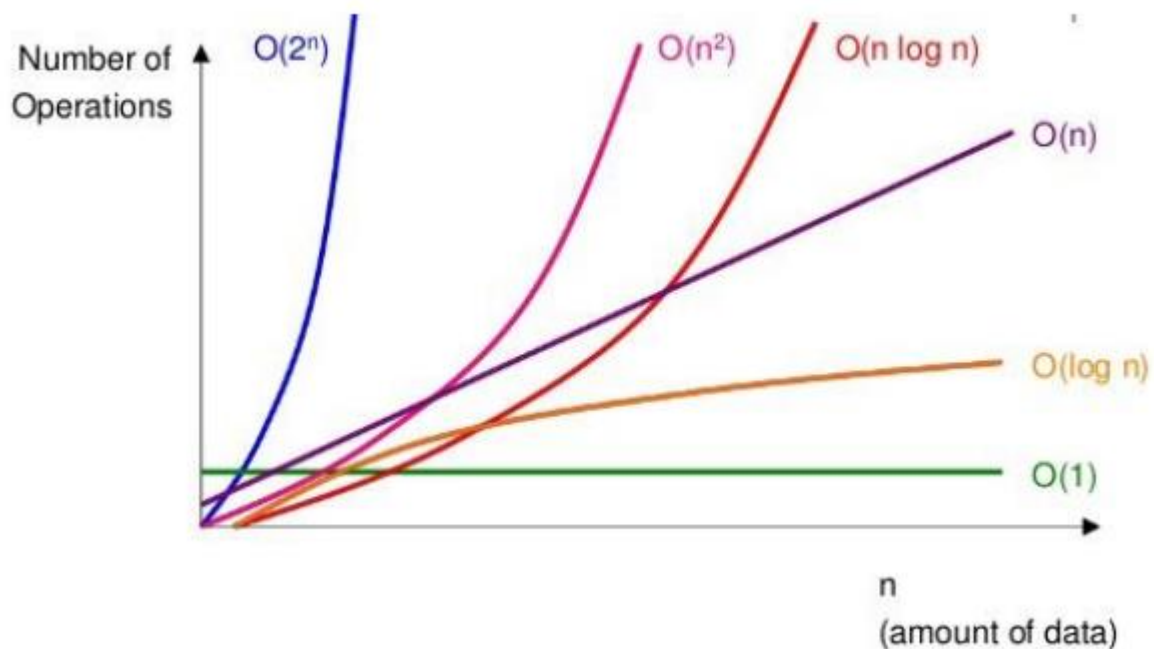
KOMPLEKSNOST ALGORITAMA

Kompleksnost algoritama s kojima radimo mjeri se funkcijama koje moramo znati napamet:

- $f(n) = 1$
- $f(n) = \log n$
- $f(n) = n$
- $f(n) = n \log n$
- $f(n) = n^2$ (ili n^3, n^4, \dots)
- $f(n) = 2^n$
- $f(n) = n!$

Strana 15

Odnos funkcija sa gornje slike izgledaju ovako:



Na ispitu ćemo morati znati raspoznati koje funkcije su najbolje, za određeni uvijet. Tablica iznad prikazuje koja je od navedenih funkcija najbrža, a koja najsporija.

ARRAY

Sa array-em kao takvim smo se upoznali još u prvom semestru na programiranju. Tada smo ga deklarirali tako što smo samo napisali vrstu podatka, te naveli količinu, i to je bilo to.

U ovom slučaju tok priče ide u potpunosti drugačije:

```
#include <iostream>
#include <array>
#include <algorithm>

using namespace std;

int main()
{
    array <int, 100> arr;

    return 0;
}
```

Koja je razlika? Razlika je u tome što je array zapravo omotač oko našeg regularnog polja, što zapravo naš array čini klasom koja dolazi s novim funkcionalnostima koje regularno polje nema.

Osnovna razlika između array-a i vectora je u tome da je array "size immutable", što znači da njemu moramo zadati neku veličinu koju nikad nećemo mijenjati.

Kako bi zapravo dobili mogućnost funkcionalnosti array-a, moramo prvo includati **<array>**, te zatim ako želimo i one funkcionalnosti koje ćemo analizirati, moramo includati i **<algorithm>**.

Uvođenjem array-a dobili smo na korištenje nekoliko novih funkcija koje će nam kasnije biti jako korisne. Sve te funkcije biti će obrađene u vectorima i jedan dio u Ishodu 2. Sintaksu za korištenje array-a nećemo pokazivati jer je sitaksa apsolutno ista kao i za obično polje, samo što se array ispod haube ponaša drugačije od regularnog polja.

VECTOR

Vector smo radili u prošlom semestru, u ovome ga radimo još detaljnije. Sada uvodimo par novih stvari što se tiče vectora.

Prije svega, vector je omotač oko polja koje može mijenjati veličinu. Elementi vectora su poslagani jedan iza drugog u memoriji (to nam govori i **.push_back()** kao optimalan način ubacivanja elemenata u vector). Vektor ima jednu manu, rezerviramo li kapacitet od 50 elemenata i probamo li ubaciti 51. element u vector alocirat će se novo, veće dinamičko polje. To je jako loše, jer se polje ne povećava uvijek za 1 element, nego čak za 1.000.000 elemenata ako se zadese uvijeti za to. **U suštini, rast vectora se dešava eksponencijalno!**

Postoje šest načina izrade vectora:

- **vector<int> jedan** – kreiranje praznog vectora
- **vector<int> dva (n)** – kreira vector od n elemenata inicijaliziranih na defaultnu vrijednost
- **vector<int> tri (n, val)** - Kreira vektor od n elemenata, svaki je kopija od val (fill)
- **vector<int> cetiri(tri.begin(), tri.end())** - Kreira vector kopiranjem elemenata iz zadanog raspona (range)
 - Prva vrijednost je početna adresa (uzima se i element na toj adresi)
 - Druga vrijednost je zadnja adresa (element na toj adresi se ne uzima)
- **vector<int> pet (tri)** - Kreira vector na način da kopira sve elemente iz zadanog vectora (copy)
- **vector<int> sest({ 11, 22, 33 })** – direktna inicijalizacija pomoću liste

Vectorima možemo provjeriti veličinu i kapacitet preko funkcija **.size()** & **.capacity()** respektivno.

Veličinu i kapacitet vektora možemo promjeniti i ručno pozivom funkcija **.resize(n)** & **.reserve(n)**.

Vektori nude nekoliko načina pristupa elementima

- **v[i]** vraća referencu na element na mjestu i
- **v.at(i)** vraća referencu na element na mjestu i
- **v.front()** vraća referencu na prvi element
- **v.back()** vraća referencu na zadnji element

Primjer rada navedenih funkcija:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

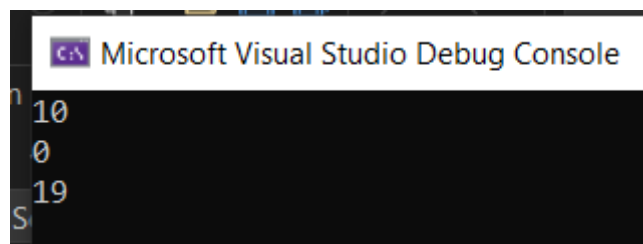
int main()
{

    vector <int> v;

    for (int i = 0; i < 20; i++)
    {
        v.push_back(i);
    }

    cout << v.at(10) << endl;
    cout << v.front() << endl;
    cout << v.back() << endl;

    return 0;
}
```



Vektori imaju i funkcije za brisanje koje glase:

- **v.clear()** – čisti cijeli vektor
- **v.erase(v.begin(), v.end())** – radi istu stvar kao i clear, samo što se koristi pointerima kako bi se očistio izabrani range. Ovom funkcijom možemo obrisati i prvih 10 elemenata tako što modificiramo funkciju da bude **v.erase(v.begin(), v.end() + 10)**.

Vektori uz funkcije za brisanje imaju i funkcije za umetanje koje će nam biti jako važne kasnije_

- **v.insert(v.begin(), i)** – insert je funkcija koja nam omogućuje da vektoru damo poziciju na koju želimo insertirati željeni element na određenu poziciju. Uz to ova funkcija dozvoljava umetanje na početak pomoću neke funkcije ili konkretno umetanje nečega preko for petlje. Insert **NE RADI** s indexima, nego isključivo s pointerima. To je ujedno i razlog zašto insertu šaljemo pointer na početak vektora.

- **v.assign()** – assign je funkcija koja nam omogućuje dodjelu / prijenos elemenata iz jednog vektora u drugi. Recimo da imamo vektore v1 i v2. Vektor v1 je ispunjen brojevima, a mi želimo prenjeti elemente iz vektora v1 u vektor v2. Jedan od načina je da pišemo for petlju i zakompliciramo stvari, a drugi je da funkciji assign koju ćemo imenovati v2.assign() u zagrade upišemo pointere na prvi i zadnji element vektora v1. Krajnji izraz izgleda ovako: **v2.assign(v.begin(), v.end())**

Kasnije ćemo obraditi i objasniti funkcije poput `emplace_back()`, `pop_front()`, `pop_back()`, `push_front()`, itd...

FOR_EACH

`for_each()` funkcija je jedna jako zanimljiva funkcija koja dolazi includana u `<algorithm>`. Ova funkcija nam služi kao svojevrsna zamjena za `for` petlju, jer njoj možemo poslati pointere na elemente u nekom range-u i uz to možemo poslati i naziv funkcije koju želimo da se izvrši za svaki taj element u range-u.

Demonstrirajmo ovo kroz jedan vrlo jednostavan primjer:

Recimo da želimo iz vektora ispisati svaki parni broj između 1 i 20. To bi odradili ovako...

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

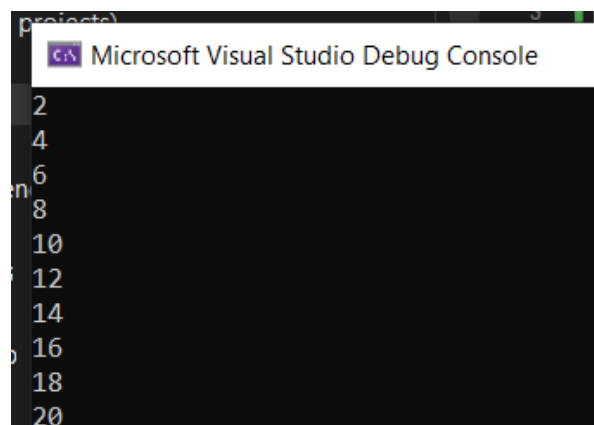
void parni(int& n)
{
    if (n % 2 == 0)
    {
        cout << n << endl;
    }
}

int main()
{
    vector<int> v;

    for (int i = 0; i < 20; i++)
    {
        v.push_back(i);
    }

    for_each(v.begin(), v.end(), parni);

    return 0;
}
```



Da sumiramo izvedbu ovog koda: kada smo pozvali `for_each` dodjelili smo mu pointer na prvi element i na zadnji element (exlusive (znaci da gleda na zadnji + 1)), te smo mu nakon toga predali pointer na funkciju koja prima referencu na `int`.

Uz `for_each` funkciju, valja napomenuti jednu funkciju koja do sada nije spomenuta. Recimo da imamo neko polje koje želimo ispisati u nazad. Logično, posežemo odmah za `for` petljom što predstavlja komplikaciju. DA bi smo se riješili komplikacija `<algorithm>` nam nudi **`reverse()`** funkciju kojoj predajemo pointere na prvi i na zadnji element.

RANDOM GENERATOR

Kako radimo RNG?

Na žalost, u C++ ne postoji funkcija koju možemo samu po sebi pozvat koja će nam generirati neke random brojeve iz nekog raspona. Valja napomenuti da zapravo i postoji, ali uvijek dobijate isti raspon random brojeva. Npr. prvi put dobijete brojeve 1, 4, 5, 7 i onda opet pokrenete program, dobit ćete opet iste brojeve.

Kako to riješavamo? Cijeli proces riješavamo tako da sami napravimo svoju funkciju koja daje random brojeve od nekog raspona. Kako ćemo to izvesti? Zapravo je vrlo jednostavno:

1. KORAK

Da bi započeli cijeli proces moramo includati zaglavlje **`<ctime>`** koje nam omogućuje korištenje funkcije **`srand`**.

2. KORAK

U mainu postavimo tzv. seed **`srand(time(nullptr))`**.

```
#include <iostream>
#include <ctime>
#include <algorithm>

using namespace std;

int main()
{
    srand(time(nullptr));

    return 0;
}
```

Valja napomenuti da ne moramo staviti **`nullptr`** unutar `time()`, compiler prihvaća i **`NULL`**.

3. KORAK

Nakon toga radimo funkciju za generiranje random brojeva.

```
int gen_rnd(int min, int max)
{
    return rand() % (max - min + 1) + min;
}
```

4. KORAK

Popunimo (u ovom slučaju) vektor s 10 random brojeva od 0 – 100, te ga isprintamo.

```
void print(int& n)
{
    cout << n << endl;
}

int main()
{
    srand(time(nullptr));

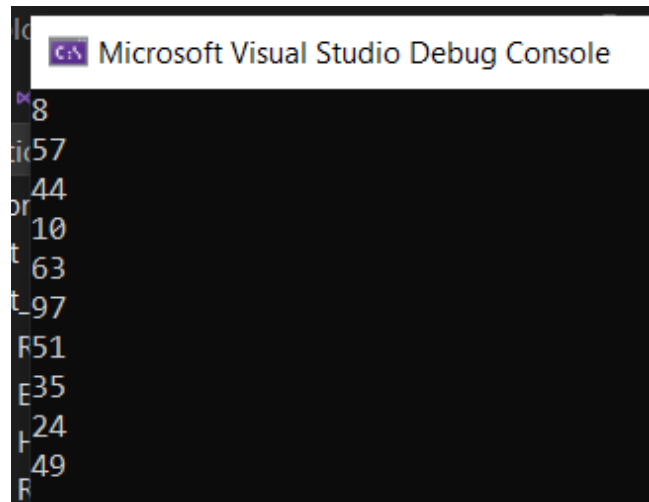
    vector<int> v;

    for (int i = 0; i < 10; i++)
    {
        v.push_back(gen_rnd(0, 100));
    }

    for_each(v.begin(), v.end(), print);

    return 0;
}
```

Rezultat programa:



ISHOD 01 – Part 3:

- File including
- Parsiranje & Template

FILE INCLUDING

U narednom periodu jako često ćemo se susretati s .txt & .csv datotekama ili vanjskim .cpp & .h datotekama. Da bi mogli raditi s tim fajlovima, iste moramo ubaciti u naš projekt.

File se includa:

Desni klik na projekt ---> Open in file explorer ---> Nađemo našu datoteku i prevučemo je u folder našeg projekta ---> Kliknemo na Show All Files ikonu koja se nalazi u alatnoj traci solution explorera u Visual Studiu ---> Desni klik na datoteku koju zelite includati ---> Include in Project

Valja napomenuti da .csv i .txt datoteke ne morate includati, možete ih samo ubaciti u projekt i prekopirati naziv u ifstream / ofstream. Datoteke .cpp & .h morate includati u projekt.

PARSIRANJE & TEMPLATE*

Odmah na samom početku, template koji će biti objašnjen naknadno nije obavezan na ispitu, ali vam olakšava život kod parsiranja za barem x100.

Parsiranje je jedan od zadataka koji Vas čeka na ispitu. Za one koji se nisu susretali sa parsiranjem datoteka, morat će dosta vježbati jer se kod mora znati napamet. Imajte na umu da je parsiranje datoteka samo primjena dosadašnjeg znanja koje ste stekli na SPA + razumijevanje funkcija koje smo do sada radili (stringstream i getline). Cijeli proces parsiranja je pun vrlo interesantnih detalja koje je potrebno dobro proučiti i provježbati. Detaljnije primjere parsiranja datoteka imate na Infoeduci, a mi ćemo proći samo jedan.

Parsiranje se sastoji od nekoliko koraka:

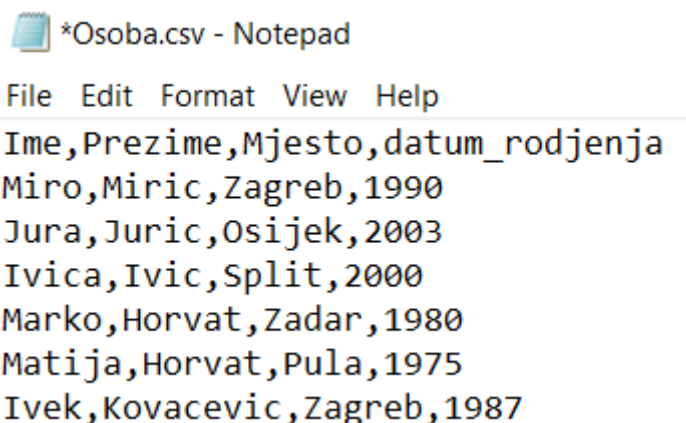
- Analiziranje datoteke (provjera s čime baratamo, tipovi datoteka, itd.)
- Ubacivanje datoteke u projekt koju želimo parsirati
- Stvaranje klase i definiranje varijabli, metoda,...
- Definiranje funkcija, stvaranje potrebnih konstruktora
- Pozivanje datoteke i provjera datoteke
- Parsiranje datoteke
- Rad s parsiranom datotekom

1. KORAK – Analiziranje datoteke

Kao primjer koristit ćemo datoteku Osoba.csv preko koje ćemo objasniti parsiranje.

Kod parsiranja s datotekama treba biti jako oprezan kako se barata istima. Jednu stvar koju ćete najčešće čuti je to da ne vjerujete izgledu fajla. Prva stvar koju morate napraviti je to da u windows exploreru uključite prikaz file extensiona. Zašto nam je to bitno? Probamo li otvoriti Osoba file sam po sebi, otvorit će nam se excel tablica. To, naravno, nije dobro po nas. Kao što se vidi iznad baratamo s .csv fajlom koji zapravo znači "Comma Seperated Values". Kao što se može vidjeti, to nije excel, nego obična tekstualna datoteka čiji su elementi međusobno odvojeni zarezom (...razmakom ili bilo kojim drugim znakom).

Kada smo riješili jedan problem koji se najčešće događa, pređimo na analizu datoteke:



```
*Osoba.csv - Notepad
File Edit Format View Help
Ime,Prezime,Mjesto,datum_rođenja
Miro,Miric,Zagreb,1990
Jura,Juric,Osijek,2003
Ivica,Ivic,Split,2000
Marko,Horvat,Zadar,1980
Matija,Horvat,Pula,1975
Ivek,Kovacevic,Zagreb,1987
```

Kao što se vidi na slici baratamo sa podacima koji su redom: Ime, Prezime, Mjesto i datum_rođenja. Sada kada malo bolje pogledamo, da bi ove podatke mogli parsirati, moramo odrediti tip podataka za svaki od stupaca. Ako bi sada išli upisati ime, prezime i mjesto direktno u visual studiu, prvo što bi napravili je to da se pozovemo na string varijable (u prijevodu, tri navedene će biti stringovi). Pratimo li logiku, datum rođenja će biti int varijabla. Super! Sada znamo da u našem kodu moramo imati tri stringa i jedan int.

2. KORAK - Ubacivanje datoteke u projekt koju želimo parsirati

Proces ubacivanja je objašnjen na početku ovog poglavlja, te je zapravo isti. Valja napomenuti da ne moramo datoteku includati u naš projekt, nego ju samo možemo copy + paste u folder u kojem se nalazi naš projekt.

3. KORAK - Stvaranje klase i definiranje varijabli, metoda,...

Nas source.cpp će biti klasičan kao i do sada, te ćemo ga za sada preskočiti.

Da bi mogli uopće mogli parsirati datoteku, trebamo imati nekakvu klasu. Klasu radimo na isti način na koji je prije objašnjeno, te definiramo private i public dijelove klase. Nakon toga se moramo prisjetiti da smo tokom analize spomenuli da imamo četiri varijable od kojih su tri stringa i jedan int.

```
#pragma once
#include <string>

using namespace std;
class Osoba
{
private:
    string ime, prezime, mjesto;
    int godina_rodjenja;
public:
    Osoba(string ime, string prezime, string mjesto, int godina_rodjenja);
};
```

Da bi osigurali da naš kod radi kako treba, moramo definirati i naš konstruktor. Kao što znamo, rekli smo da je defaultni konstruktor jako loš za nas jer ne inicijalizira varijable, nego u njih stavlja smeće. Korisnički definiran konstruktor će nam biti potreban kada budemo morali baratati s klasom.

Super! Naš header je sada spreman, te možemo preći na definiranje.

4. KORAK - Definiranje funkcija, stvaranje potrebnih konstruktora

```
#include "Osoba.h"

Osoba::Osoba(string ime, string prezime, string mjesto, int godina_rodjenja)
{
    this->ime = ime;
    this->prezime = prezime;
    this->mjesto = mjesto;
    this->godina_rodjenja = godina_rodjenja;
}
```

Ovo nam je za sada jedina metoda/konstruktor koji se nalazi u Osoba.cpp fajlu. Detaljniji rad nad datotekama biti će objašnjen na vježbama, te ćete tamo proći puno detaljnije primjere. Radi jednostavnosti ove skripte, ovdje će biti objašnjen samo osnovni rad nad fajlom i algoritam parsiranja.

Kada smo definirali naš konstruktor i sve dodatne metode koje su nam potrebne (ako postoje), možemo preći na naš source.cpp file.

5. KORAK - Pozivanje datoteke i provjera datoteke

Datoteku koju smo maloprije ubacili u naš projekt moramo nekako i pozvati i provjeriti da li ga program može pronaći. To radimo u mainu na način kojim smo radili u prvom semestru na programiranju. Ovaj dio koda je uvijek isti:

```
ifstream in("Osoba.csv");  
  
if (!in)  
{  
    cout << "Nope!" << endl;  
    return 1;  
}
```

Naravno, uz ovaj komad koda moramo i includati **<fstream>**.

6. KORAK - Parsiranje datoteke

7. KORAK - Rad s parsiranom datotekom

ISHOD 02:

- Interatori
- List, Forward List
- Stack
- Queue
- Adaptive Containers

INTERATORI

LIST & FORWARD LIST

STACK

QUEUE

ADAPTIVE CONTAINER