Malo povijesti

- Simula (1967.)
  - Prvi programski jezik sa svojstvima objektno-orijentirane paradigme
  - Namijenjen izgradnji sustava za simulaciju
  - Uveden pojam klase
- Smalltalk (1972.)
  - Prvi “pravi” (čisti) objektno-orijentiran programski jezik (“sve je objekt”)
  - Razvijen u Xerox PARC laboratoriju
  - Najkorištenija verzija je Smalltalk-80
- C++
  - “hibridni” objektno-orijentirani jezik nastao iz C-a - ispočetka se zvao “C s klasama” (“*C with Classes*”)
  - Razvio ga je Bjarne Stroustrup (1983.) u Bell Labs
  - Inicijalna ANSI standardizacija je dovršena (tek) 1998., a 2003. je izdana standardna verzija s ispravljenim pogreškama
  - C++ je “predak” danas široko korištenih jezika - Java, C# i VB.NET

Upoznavanje s konceptima OOP

- Krećemo od jednostavnog problema koji ćemo riješiti u C- u
- Identificirat ćemo nedostatke tog rješenja i kroz postupno uvođenje koncepata objektne orijentacije vidjeti kako se oni mogu riješiti implementacijom u C++

Problem kojim ćemo se baviti

- Zadana je datoteka u kojoj se nalazi određeni (nepoznat) broj podataka. Radi jednostavnosti, datoteka je sekvencijalna, formatirana i sadrži u svakom retku samo jedan podatak cjelobrojnog tipa
- Potrebno je napisati program koji će učitati podatke iz datoteke u memoriju (u strukturu polja – nakon učitavanja možemo dohvatiti podatke po indeksu) i omogućiti njihovu obradu

- VAŽNA NAPOMENA
  - S obzirom da podaci moraju **ostati** u memoriji nakon učitavanja, nije moguće jednostavno rješenje slijednog čitanja podataka (jedan po jedan) sve do kraja datoteke!

Jednostavno C rješenje

- Pročitamo sve podatke iz datoteke u jednom prolazu radi utvrđivanja koliko ih ukupno ima
- Zatim alociramo polje potrebne veličine pomoću *malloc*
- Ponovno prolazimo kroz datoteku i iznova učitavamo podatke (s time da ih sada spremamo u alocirano polje)
- NAPOMENA
  - Ovaj pristup zahtijeva dva prolaza kroz datoteku i primjenjiv je kada efikasnost programa (vrijeme izvršavanja) nije problem

Bolje rješenje: Alociranje memorije po potrebi

- Možemo koristiti funkciju **`realoc`**
  - Alociramo inicijalni blok memorije (npr. za 100 podataka) i zatim ga po potrebi povećavamo tijekom učitavanja
- Sada imamo 2 veličine koje “opisuju” učitane podatke
  - Pokazivač na alociranu memoriju
  - Količinu alocirane memorije
- Dobro je “povezati” podatke – definiramo strukturu

```
struct DinamickoPolje
{
  int *Podaci;
  int BrojElem;
};
```

Skup funkcija koje rada sa strukturom DinamickoPolje

- Funkcije za strukturu DinamickoPolje:
  - `int Inicijaliziraj (`
    `struct DinamickoPolje *Polje,`
    `int InicijalniBrojElem);`
  - `void Izbrisi (`
    `struct DinamickoPolje *Polje);`
  - `int PostaviNovuVelicinu (`
    `struct DinamickoPolje *Polje,`
    `int NoviBrojEl);`
- (analogija s funkcijama za strukturu FILE (struct FILE): fopen, fscanf, fread, ...)

Primjer P01

- DinamickoPolje_C_osnovna_impl
- Podaci (ulazna datoteka)

Tehnički nedostaci C implementacije:

- Sami moramo paziti na iskorištenost prostora (`BrojUcitanih`)
  - ova struktura je samo vrlo jednostavan “omotač” (engl. *wrapper*) oko pokazivača („sakrili“ smo `malloc` i `realloc` od korisnika strukture, ali nismo definirali nikakvo dodatno ponašanje)
- Pristup podacima ide preko direktnog korištenja pokazivača koji je dio strukture (`Polje.Podaci[ ]`)

Poboljšanja:

- Primjena funkcija **`get`** i **`set`** za dohvaćanje i postavljanje vrijednosti elemenata u polju
- Dodavanje podatka o iskorištenosti alociranog prostora u strukturu **`DinamickoPolje`**
- Definiranje dodatnih funkcija za rad sa strukturom **`DinamickoPolje`**

Proširenje strukture DinamickoPolje

```
struct DinamickoPolje
{
  int *Podaci;
  int BrojElem;
     // stvarni broj elemenata u polju
  int MaxBrojElemenata;
     // maksimalno raspoloživi prostor
};
```

Funkcije potrebne za rad s dinamičkim poljem

Inicijaliziraj

Izbrisi

PostaviNovuVelicinu

PostaviElement

DodajElementNaKraj

DohvatiElement

BrojElemenata

Prototipovi funkcija

```
int  Inicijaliziraj (struct DinamickoPolje *Polje, int
   MaxBrojElem);

void Izbrisi (struct DinamickoPolje *Polje);

int  PostaviNovuVelicinu (
   struct DinamickoPolje *Polje, int NoviBrojElem);

void PostaviElement (
   struct DinamickoPolje *Polje, int Ind, int Vrijednost);

int DodajElementNaKraj (struct DinamickoPolje *Polje, int
   Vrijednost);

int DohvatiElement (struct DinamickoPolje *Polje, int
   Indeks);

int BrojElemenata (struct DinamickoPolje *Polje);
```

Problem dojave pogreške

- Funkciju **`DohvatiElement()`** smo deklarirali kao

```
int DohvatiElement (struct
  DinamickoPolje *Polje, int Indeks);
```

- A što ukoliko se zatraži vrijednost elementa polja za nepostojeći indeks ?
  - Potrebno je nekako naznačiti pogrešku !

Signaliziranje pogreške preko povratnog argumenta funkcije

- Novi prototip:

```
int DohvatiElement2 (
  struct DinamickoPolje *Polje,
  int Indeks,
  int *DohvacenaVrijednost);
```

- Preko povratnog parametra (*return*) signaliziramo pogrešku (ako je bude)
- Preko call by reference vraćamo dohvaćenu vrijednost
- Alternativno: Izvesti naredbu *exit*() pa neka programer koji koristi DinamickoPolje popravi pogrešku u svojem kodu

Primjer P02

- P02_DinamickoPolje_C_get_set

- P01 DinamickoPolje_C_osnovna_impl
- Podaci (ulazna datoteka)

Ozbiljan nedostatak !

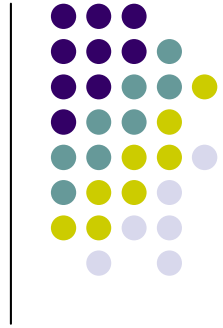
- Naše DinamickoPolje se može iskoristiti samo za *int* podatke !
  - Što ukoliko imamo datoteku u kojoj su zapisani *float* podaci ?
- Kako poopćiti našu implementaciju strukture DinamickoPolje da se može iskoristiti za bilo koji tip podatka ?
  - Klasično C rješenje – korištenje *typedef*-a
  - Bolje (i složenije) rješenje – koristimo void * kao tip podataka u polju
    - Programer kod iskorištavanja dinamičkog polja mora koristiti *cast* operatore!

Primjer: P02_DinamickaPolja_C_typedef

Nedostaci sa stajališta dizajna programa

- Svakoj funkciji moramo prenositi pokazivač na strukturu
  - Ovo je primarno sintaksni “nedostatak”
  - C (ali i Pascal, Fortran, Basic) programeri s takvom paradigmom “žive” već 20-ak godina ☺
- Pozivi funkcija iz biblioteke su isprepleteni s pozivima drugih funkcija i sintaksno izgledaju isto
  - Smanjuje se razumljivost programa

Nedostaci – dio drugi



- Mogućnost sukoba imena (engl. *name clash)*
  - Što ukoliko je u nekoj drugoj biblioteci definirana funkcija s istim imenom ?
  - Koristimo hrvatske nazive za varijable i funkcije pa i nije toliki problem !
    - Kod korištenja engleskih termina (kod izrade biblioteka funkcija bitan zahtjev ukoliko se želi omogućiti široko/internacionalno korištenje razvijene biblioteke) je situacija značajno gora (funkcije: *initialize, setSize, cleanup* ili *delete* !!!)
- Što ukoliko netko izravno promijeni vrijednost podatka u strukturi!
  - Definirane funkcije se oslanjaju na činjenicu da