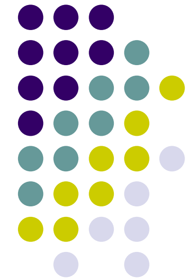
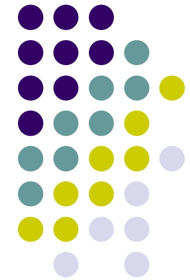
Klasa



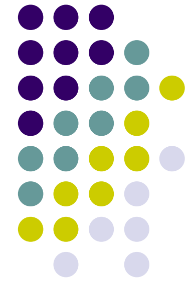
- Uvodimo ključnu riječ **`class`**
  - Klasa poopćava pojam strukture iz C-a
- Razlika između klase (class) i strukture (struct) u jeziku C++
  - Sa stanovišta C++ prevoditelja primarno sintaksna – kod strukture je sve podrazumijevano **`public`** dok je kod razreda sve podrazumijevano **`private`**
- Konceptualna razlika:
  - Struktura je (ipak) namijenjena modeliranju skupa jednostavnih podataka nad kojima ostali dijelovi programa direktno operiraju
  - Klasa kao primarni koncept OO paradigme (prisutan u svim OO jezicima!) je namijenjen modeliranju (predstavljanju) koncepata iz područja problema koji rješavamo (**`DinamickoPolje`**, **`Stog`**, **`HashFile`**, ...) koji imaju <u>složeno ponašanje</u> realizirano preko skupa članskih funkcija

Primjer:
P05_DinamickoPolje_Cpp2_prava_pristupa



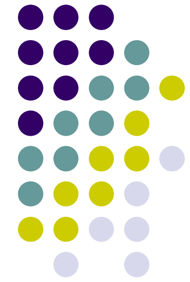
```
class DinamickoPolje
{
public:
   int  Inicijaliziraj(int inMaxBrojElem);
   void Izbrisi();
   int  PostaviNovuVelicinu(int NoviBrojElem);

   void PostaviElement(int Indeks, int Vrijednost);
   int  DohvatiElement(int Indeks);
   int  DodajElementNaKraj(int Vrijednost);

   int  BrojElemenata();

private:
   int  *_Podaci;

   int  BrojElem; // koliko stvarno ima elemenata u polju
   int  MaxBrojElemenata; // maksimalni raspoloživi prostor
};
```

Apstrakcija i enkapsulacija – osnovni elementi OO paradigme



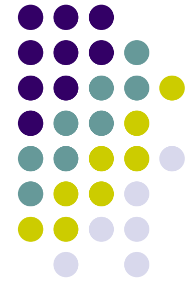
- Apstrakcija – odbacivanje onoga što je sporedno, posebno i slučajno radi onoga što je opće, zakonito i bitno
- Enkapsulacija (učahurivanje) - onemogućavanje pristupa varijablama klase osim putem ugrađenih metoda za njihovo čitanje i pisanje. Time se osigurava da objekt ne može doći u neko nepredviđeno stanje iz bilo kojeg razloga

A & E (nastavak) Apstrakcija



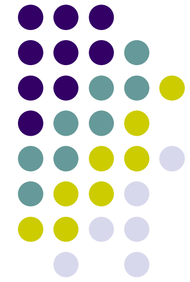
- Apstrakcija – klase / objekti predstavljaju koncepte iz domene problema koji rješavamo
- Klasa `DinamickoPolje`
  - Apstrakcija koncepta dinamičkog polja s dobro definiranim karakteristikama
- Nije li to isto i struktura u C-u?
  - Struktura predstavlja agregatni skup podataka nad kojima operiraju “vanjski” elementi programa (funkcije)
  - Vanjske funkcije nisu dio strukture – zbog toga struktura nije “potpuna” jer je za razumijevanje koncepta koji predstavlja potrebno proučiti nešto što nije dio same definicije strukture
  - Struktura nije “zatvorena” u smislu da ne upravlja sama svojim ponašanjem i stanjem!
  - Modeliranjem koncepta pomoću klase rješavamo navedene probleme

A & E (nastavak)
Enkapsulacija



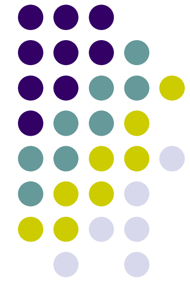
- Enkapsulacija
  - “niti jedan dio složenog sustava ne bi smio ovisiti o unutrašnjim detaljima drugog dijela”
  - “unutrašnji detalji” – korisnika klase `DinamickoPolje` uopće ne treba zanimati kako je realizirana njegova funkcionalnost

Primjer DinamickoPolje implementacija



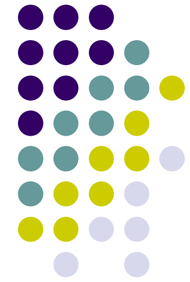
- Koncept dinamičkog polja je vrlo jednostavan i praktički implicira način implementacije (dinamička alokacija memorije za elemente polja), ali kod složenijih klasa to ni u kojem slučaju nije tako
- Čak i kod dinamičkog polja možemo imati varijabilnost u implementaciji:
  Npr., ako je rukovanje memorijom skriveno od korisnika klase, može se izraditi vlastiti modul za upravljanje memorijom (ne koristimo **`malloc`** i **`realloc`**, već vlastitu implementaciju – npr. radi efikasnosti)

Primjer DinamickoPolje enkapsulacija



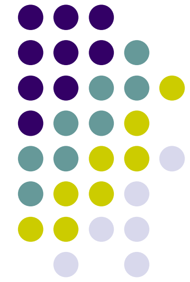
- Enkapsulaciju postižemo deklariranjem unutrašnjih detalja razreda kao **`private`**
- Budući da tada ionako ne može pristupiti tim dijelovima, korisnik klase **`DinamickoPolje`** ne može o njima ni ovisiti (zato jer ih ne može izravno referencirati u programskom kodu kojega on piše)

Javno sučelje klase



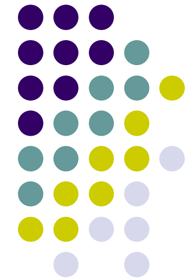
- Sve što je u klasi deklarirano kao `public` dio je javnog sučelja klase
- Javno sučelje klase predstavlja “prozor u svijet” kroz koji razred komunicira s ostalim dijelovima programa
- Definiranjem članskih varijabli kao `private`, klasa “skriva” svoje stanje

Enkapsulacija u stvarnom životu i u programiranju



- Svaki sustav kod kojega nije bitno **kako** radi već **što** radi
- Postoji sučelje putem kojega korisnik komunicira sa sustavom (npr. daljinski upravljač - Play, Stop, Eject, ...)
- Za korisnika klase `DinamickoPolje` je bitno da klasa ima **očekivano ponašanje**, a kako je unutar klase omogućeno takvo ponašanje, korisnika (uglavnom) **ne zanima**!

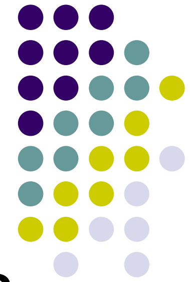
Pojam objekta



- Ovako izgleda **deklaracija** klase (novi tip podatka):

```
class MojaKlasa {
    ...
};
```

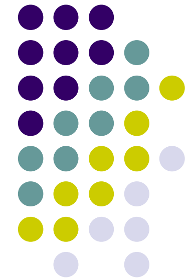
- Slično kao i kod struktura, klasa predstavlja **predložak** iz kojega će se kreirati **objekti**
  - Kod struktura se konkretna instanca strukture naziva **varijabla strukture**
  - Kod klase se konkretna instanca klase naziva se naziva **objekt**
- Klasa je jedna, a iz nje se može instancirati proizvoljan broj objekata
  - Svi objekti imaju isti skup članskih varijabli i članskih funkcija
  - Objekti se razlikuju po vrijednostima koje imaju njihove članske varijable (te vrijednosti predstavljaju **stanje objekta**)
- Analogija sa strukturom polje – svi članovi su istog tipa, a vrijednosti im se razlikuju za svaki indeks polja



Kreiranje objekata

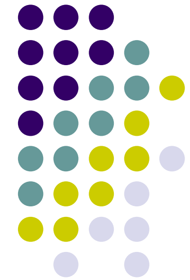
- Slično kao i kod struktura, kreiranje objekta primarno podrazumijeva alociranje prostora u memoriji gdje će objekt (odnosno njegove članske varijable) biti smješten
- Moguća su dva standardna načina:
  - Smještanje objekta na stog - Objekt se deklarira kao lokalni objekt unutar funkcije
  - Smještanje objekta na *heap (*gomilu*)* - "životni vijek" objekta nije vezan uz kontekst izvođenja funkcije već se objekt eksplicitno mora "uništiti" (izbrisati iz memorije)

Heap - gomila



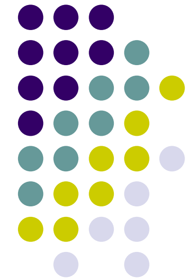
- Za potrebe rada s objektima na *heap*-u, uvedena su dva nova operatora:
  - `new` – operator za kreiranje objekata *na heap*-u
  - `delete` – operator za brisanje objekata s *heap*-a

Primjer za new i delete



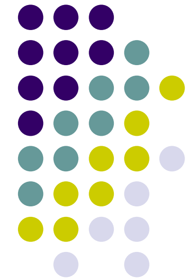
```
class MojaKlasa
{
public:
   int _MojPodatak;
};

int main(int argc, char* argv[])
{
   MojaKlasa objStog; // objekt na stogu
   objStog._MojPodatak = 10;

   MojaKlasa *pStog = new MojaKlasa(); // objekt na heap-u
   pStog->_MojPodatak = 10;

   delete pStog; // moramo eksplicitno osloboditi memoriju

   return 0;
   // po završetku funkcije, objStog će se automatski
   ukloniti iz memorije
}
```

Još o new i delete



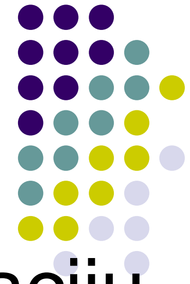
- *new i delete* nisu namijenjeni isključivo za kreiranje i uništavanje objekata:
- Predstavljaju općenitu zamjenu za *malloc* i *realloc*
- *Type safe* verzija – točno se zna za kakav tip podatka se alocira memorija

Primjer: P06_Primjer_new_delete



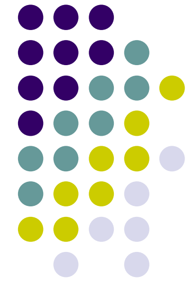
```
float  *pFloat = new float;
int    *pInt = new int[10];
char   *pString = new char[20];

delete      pFloat;
delete      [] pInt;
delete      [] pString;
```

Za brisanje **polja** mora se koristiti operator `delete []`



- Kreiranje objekta ipak ne znači **samo** alokaciju memorije za smještanje objekta !
  - Bitno je u kakvom **stanju** se objekt nalazi nakon kreiranja, odnosno kakve su mu vrijednosti članskih varijabli - problem **inicijalizacije** !
- Javlja se i u C-u:
  - Nakon deklaracije `int a`; nije jednoznačno definirano kakvu vrijednost ima varijabla `a`
- Kod objekata je stvar još složenija jer mogu imati više članskih varijabli
  - Kako inicijalizirati **pokazivače** koji su dio klase ?!

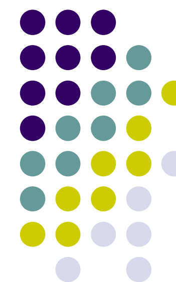
Inicijaliziranje objekta



- Rješavanju ovog problema kod klase `DinamickoPolje` je namijenjena funkcija `Inicijaliziraj()` koja dovodi kreirani objekt u ispravno stanje

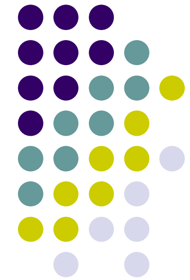
- Što ako kreiramo objekt i zaboravimo pozvati funkciju `Inicijaliziraj()` ?
  - Dolazi do pogreške kod korištenja objekta

Sljedeći put



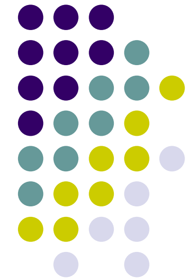
- Konstruktori

Pojam konstruktora objekta/klase

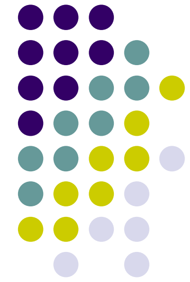


- Specijalna članska funkcija namijenjena inicijalizaciji stanja objekta kod njegovog kreiranja
- Prepoznaje se po imenu funkcije – **mora** biti isto kao i ime razreda

```
class MojaKlasa {
    public:
    MojaKlasa() { ... }     //
      konstruktor bez parametara
    MojaKlasa(int a) { ... }     //
      konstruktor s parametrom
  };
```



- Primjer *overloading*-a (preopterećenja) funkcije
  - Imamo funkcije **istog imena** (u C-u nije dozvoljeno) a prevoditelj ih razlikuje po **parametrima** !
- Konstruktor nema povratnog parametra – ne vraća i ne može vratiti nikakav podatak nakon izvršavanja !
  - A ako dođe do pogreške koju treba signalizirati ostatku programa – treba “baciti” izuzetak (engl. *exception*)



- Primjer korištenja:

```
void main() {
    MojaKlasa     a;
    MojaKlasa     b(10);
    MojaKlasa     *c = new MojaKlasa();
    MojaKlasa     *d = new MojaKlasa(10);
}
```

- A kako su onda radili naši prethodni primjeri (bez definiranog konstruktora) ?
- Prevodilac za svaki razred za koji nije eksplicitno definiran konstruktor sam dodaje podrazumijevani (engl. *default*) konstruktor