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

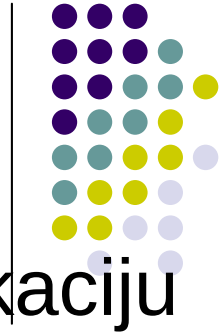
- **`new`** i **`delete`** nisu namijenjeni isključivo za kreiranje i uništavanje objekata:
- Predstavljaju općenitu zamjenu za **`malloc`** i **`realloc`**
- *Type safe* verzija – točno se zna za kakav tip podatka se alocira memorija

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 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

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 klase

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

- Primjer *overloading*-a (preopterećenja, preklapanje) funkcije
  - Imamo funkcije **istog imena** (u C-u nije dozvoljeno) a prevoditelj ih razlikuje po tipu **parametara**
- 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 koristiti iznimku (engl. *exception*)

- Primjer korištenja:

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

Kako su radili prethodni primjeri (bez definiranog konstruktora) ?

- Prevoditelj za svaku klasu za koju nije eksplicitno definiran konstruktor sam dodaje podrazumijevani (engl. *default*) konstruktor
  - Konstruktor bez parametara koji članske varijable inicijalizira na neke podrazumijevane vrijednosti

Destruktori

Destruktori

- Definiranjem konstruktora smo “pokrili” kreiranje objekta, a što je s brisanjem (uništavanjem) ?
- Možemo definirati **destruktor** klase
  - Članska funkcija koja će se pozivati prilikom uništavanja objekta
- “Uništavanje” objekta znači brisanje objekta iz memorije
  - Tehničkim žargonom – objekt izlazi iz *dosega* (engl. *scope*)
  - Za objekte kreirane na stogu – trenutak kada funkcija u kojoj su deklarirani završava i briše sve svoje podatke sa stoga
  - Za objekte kreirane na *heap*-u (gomili)– trenutak kad se nad njima poziva *delete*
- Ukoliko je u klasi definiran destruktor, prevoditelj će ga automatski pozvati u trenutku uništavanja objekta

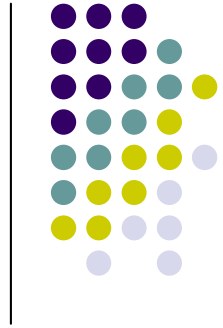
P06_PrimjerKonstruktorDestruktor

```
class MojaKlasa {
    public:
  MojaKlasa() {...}            // konstruktor 1
  MojaKlasa(int a) {...}       // konstruktor 2
  ~MojaKlasa() {...} // destruktor (jedini)
};
void main() {
  MojaKlasa  a;
  MojaKlasa  *b = new MojaKlasa();
  ... // tijelo funkcije
  delete b; // eksplicitni poziv destruktora
  }
  // destruktor za a se poziva
  // prije izlaska iz funkcije
```

Kako su radili prethodni primjeri (bez definiranog destruktora) ?

- Uništavanje objekta je podrazumijevalo samo oslobađanje memorije (alocirane za smještaj samog objekta !) i to za to se pobrinuo prevoditelj
- Destruktor je “glumila” funkcija Izbrisi() koja je oslobađala zauzetu memoriju
- A što da smo je zaboravili pozvati ?
  - Imamo *memory leak* (curenje memorije) u programu – zauzeli smo resurse računala ali ih nismo oslobodili iako ih više ne koristimo !
  - “zaboravnost” programera u takvim slučajevima može voditi izrazito nezgodnim pogreškama

Prednost destruktora



- Prevoditelj se brine da se pozove funkcija za uništavanje objekta (destruktor)
- Vrijedi samo za objekte kreirane na stogu
- Kod objekata kreiranih na *heap*-u (pomoću *new*) **mora se pozvati *delete*** !
  - Kreator objekta određuje kada će se on uništiti
- Primjer:
  P06_DinamickoPolje_Cpp3_konstruktori