Objekti sa zauzetim resursima

- Naravno, ove zauzete resurse treba propisno i osloboditi (dealocirati) kada objekat prestaje da živi. Ako se predviđa korišćenje ovih objekata po vrednosti (svih kategorija životnog veka), odnosno ugradnjom (kao podobjekti, npr. objekti članovi), njihovo uništavanje je implicitno. Na primer:

```
int main () {
  string str("Hello");
  …
}
```

Ovde se završava životni vek objekta *str*

- Prema već navedenim pravilima, ako se u klasi ne navede eksplicitno destruktor, prevodilac će generisati implicitni destruktor koji vrši destrukciju objekata članova pozivom njihovih destruktora; međutim, za članove koji su ugrađenih tipova, destrukcija nema nikakvog efekta, što je ovde slučaj, pošto je član *str* pokazivač. Prema tome, u ovom slučaju, dinamički niz znakova neće biti dealociran, pa će postojati problem curenja memorije (*memory leak*)

- Zato nam je ovde neophodan korisnički definisan destruktor koji vrši potrebnu dealokaciju:

```
class string {
public:
  string () : str(nullptr) {}
  string (const char*);

 ~string () { delete [] str; str = nullptr; }
  …

};
```

Objekti sa zauzetim resursima

- Dalje, ako se predviđa korišćenje ovih objekata po vrednosti (svih kategorija životnog veka), korisnici ove klase očekivaće i mogućnost inicijalizacije kopiranjem. Na primer:

```
int main () {
  string s1("Hello"), s2(s1);
  …
}
```

- Prema već navedenim pravilima, ako se u klasi ne navede eksplicitno konstruktor kopije, prevodilac će generisati implicitni konstruktor kopije koji vrši inicijalizaciju kopiranjem objekata članova pozivom njihovih konstrukora kopije; međutim, za članove koji su ugrađenih tipova, vrši se prosto kopiranje vrednosti, što je ovde slučaj, pošto je član *str* pokazivač. Prema tome, u ovom slučaju, objekti klase *string* biće *plitko kopirani* (*shallow copy*), što znači da će se kopirati samo pokazivač, a ne i dinamički niz na koji on ukazuje
- Ovo nije željeno ponašanje, jer će dovesti do toga da kopirani objekti (npr. *s1* i *s2* u primeru gore) dele isti dinamički niz, pa će promene učinjene u jednom biti vidljive i u drugom. Ovde je namera da ti objekti postoje kao nezavisni entiteti i da se njihove izmene rade nezavisno
- Zato nam je ovde neophodan korisnički definisan konstruktor kopije koji vrši *duboko kopiranje* (*deep copy*):

```
class string {
public:
  string () : str(nullptr) {}
  string (const char*);
  string (const string& s) : string(s.str) {}

 ~string () { delete [] str; str = nullptr; }
  …

};
```