OOP TUTORIAL

Kako su materijali vezani uz OOP poprilično loši, a stanje sa poznavanjem OOP-a od strane mnogih koji su prisustvovali mass instrukcijama jako loše, odlučio sam pokušati sa novim tutorial-om. Nadam se da ću uspjeti napisati nešto kvalitetno i razumljivo, jer stvarno bi bila šteta ne upoznati nešto što se danas masovno koristi u realnom svijetu (koliko god software development može biti realan svijet ;)).

Dokumenti:

- Rješeni zadatci iz OOP-a (Monikina zbirka):
  http://free-zd.t-com.hr/ivan444/faks/zbirka2_3oop.pdf
- Rješeni zadatci iz stoga (Monikina zbirka):

  http://free-zd.t-com.hr/ivan444/faks/zbirka2_2stog.pdf

Najveći problem je što nema nekog normalnog materijala za učiti OOP... ako u google upišete OOP (ili značenje te kratice) dobit će te tisuće tekstova, ali ni jedan neće biti nama primjeren, jer mi zapravo nismo radili ni osnove OOP-a... Ja ću se pokušat zadržat samo na nama potrebnim stvarima, a sve što nismo učili spomenut u max 2-3 riječi (ako baš moram spomenut).
Ovo neće biti neko kvalitetno "djelo" koje može stajat samostalno, već se od vas očekuje da prvo prođete slide-ove...

<manje bitan dio; slobodno preskoči>
Uvod: Što je OOP? Čemu to? I ima li uopće smisla ili im je samo falilo gradiva za kolegij?
Objektno orijentirano programiranje je jedan način programiranja (programska paradigma) koji se danas masovno koristi za razvoj software-a u pravom (poslovnom) svijetu.
Prava nadmoć OOP-a nad imperativnom paradigmom (način na koji smo dosad pisali programe) je u organizaciji i zaštiti koda prilagođenoj radu u većim timovima.
Vi sad programirate na način da napišete klasu koju će vaši kolege moći koristiti služeći se samo javnim sučeljem i komentarima javnih funkcija i varijabli vaše klase. Ukoliko se javi potreba za poboljšanjem te klase, vaš kolega može uzet vašu klasu i samo obaviti nasljeđivanje i dodati ono što njemu treba. Na taj način ne mijenja vaš kod i dobiva dvije nove klase koje su u određenom odnosu, ali ovo je za nas (zasad) sasvim nebitno...
I, još jedna od važnijih stvari je reuseability koda (mogućnost ponovnog korištenja koda). Jer, nakon što napišete jednu klasu vi je možete ponovo koristiti u nekom drugom projektu bez ikakve izmjene (jer dobro napisana klasa se brine o ispravnosti unesenih parametara i trebala bi biti NEOVISNA o okolini), a ukoliko je izmjena potrebna, vi nećete mijenjati postojeću klasu, već obaviti nasljeđivanje (što nismo radili na ASP-u...).
Krenimo sa nama bitnim stvarima (ilitiga potrebnim za prolaz ASP-a :)).
<Kraj manje bitnog dijela>

1. Reference

(Reference nisu nešto čiesto vezano uz OOP, ali trebat će nam za neke stvari, pa da ih objasnim)

Reference su nešto kao kopije varijabli, ali ne kopiraju memoriju, već samo referencu na taj memorijski prostor (kao neka vrsta pokazivača).

Znači, kad stvorite referencu, s njom radite kao da je to vaša stara varijabla.
Ostatak objašnjena ide preko koda i komentara.
Osnovni prikaz korištenja referenci.

Kod:

```
        int a = 5;
        int *p = &a;
        int &ref = a;   // PAZI! Ovo je samo notacija! Nemojte ovo
shvatiti doslovno!

        printf("%d, %d, %d", a, *p, ref);
        //      Ispisuje se: 5, 5, 5

        printf("%p, %p, %p", &a, p, &ref);
        //      (%p je format za ispis adrese varijable)
        //      Ispisuju se 3 iste adrese

        *p = 9;
        printf("%d, %d, %d", a, *p, ref);
        //      Ispisuje se: 9, 9, 9

        ref = 7;
        printf("%d, %d, %d", a, *p, ref);
        //      Ispisuje se: 7, 7, 7

        /*      Znači, ista stvar je koristili
                varijablu "a" ili "ref".
                Sad nam je ista stvar napisali npr.:
                a = a + 444;
                ili
                ref = ref + 444;
                Operacija mijenja jednu memorijsku lokaciju,
                ali prikaz te lokacije se može dobiti pomoću
                2 različite varijable, tj. "a" i "ref" (i "*p")
                u našem slučaju.
        */
```

Primjera rada sa referencama i funkcijama (obrati pažnju na ispis!):

Kod:

```
void f(int &ref) {
        ref = 100;
}

int main() {
        int a = 5;

        f(a);
        printf("%d", a);
        //      Ispisuje se: 100

        return 0;
}
```

2. Klasa, konstruktori, destruktori

Klasu možete gledati kao nekakav skup podataka sa određenim pravima pristupa i predefiniranim operacijama (najkraće i najzamršenije rečeno :)).
Ona je nekakav prototip/nacrt objekta (objekt je fizička realizacija klase (instanca), tj. zauzima fizički prostor u memoriji).

Privatne i javne varijable i funkcije:
Najbitnija stvar koju morate znati o objektima jest da su objekti SAMOSTALNE CJELINE. Svaki objekt za svoje podatke koristi svoju memoriju, i u većini slučajeva ne dopušta direktnu izmjenu svojih resursa od strane okoline, već se to radi preko članskih funkcija (više o tome u dijelu sa getterima i setterima).
Takav način organizacije nam omogućavaju modifikatori pristupa (private, public, i manje bitni).

Znači:
Kod:

```
class ModifikatoriPristupa {
        private:
                int x;
        public:
                int y;

                void setX(int x) {
                        this->x = x;
                }
                int getX() {
                        return this->x;
                }
                void zoviIspis() {
                        ispisiX();
                }
        private:
                void ispisiX() {
                        cout << x;
                }
};      // Primijetite da definicija klase završava sa ";".

int main() {
        ModifikatoriPristupa mod;

        // slijedećih 5 linija radi
        mod.setX(12);
        cout << mod.getX();
        mod.y = 15;
        cout << mod.y;
        mod.zoviIspis();

        // slijedeće 3 linije javljaju compile error
        mod.x = 4;
        cout << mod.x;
        mod.ispisiX();

        return 0;
}
```

Referenca "this" (ili bolje rečeno "pokazivač"):
Njome označavamo da želimo raditi sa članskom varijablom klase.
"this->x" možete čitati kao "moj x"
U slučaju kao kod funkcija getX() i ispisiX() nije nam potrebna referenca na člansku varijablu, ali kod funkcije setX(int x) nam je potrebna. Stvar je u tome što ukoliko vi želite očitati varijablu x unutar neke funkcije, compiler će prvo tražiti tu varijablu na lokalnoj razini (znači, varijablu stvorenu unutar funkcije ili predatu preko argumenta), a tek ako je ne nađe, kreće u potragu na višoj razini ("globalnijoj").
Meni ta referenca čini kod čitljivijim (jer odmah se vidi radi li se o članskoj varijabli ili nekoj drugoj), i smanjuje mogućnost greške kad imamo veće kodove.

Zatim, operator "->". Ovo je jednostavno... budući da je "this" pokazivač na strukturu, mi možemo pisati:
(*this).var1
ali i:
this->var1
tj.
"->" je samo druga notacija za "(*).". (čisto radi urednosti)

Scope operator "::":
Operator "vidljivosti". Služi nam da članske funkcije (funkcije koje pripadaju klasi) izvučemo van nje radi preglednosti koda (time nam u klasi ostanu samo prototipovi funkcija). Iako, mislim da je to sasvim nepotrebno, jer, ako imate gotovu klasu, rijetko kad će te ići gledati njen kod (jer to i je ideja OOP-a, napraviš klasu i nakon toga je koristiš preko njenog javnog sučelja, skoro bez razmišljanja), a masa editora nakon što napišete "ImeKlase." vam izlista sve njene funkcije i varijable, te prikaže što možete, a što ne možete koristiti (javno/privatno (zaštićeno)).
Tako da na scope operator možete zaboraviti.

Konstruktori:
Kad se objekt stvara (o stvaranju objekta u sljedećem dijelu), prvo se zauzima memorija za njene varijable, a nakon toga poziva konstruktor.
Konstruktor je funkcija koja inicijalizira resurse objekta na neke standardne ili argumentom prenesene vrijednosti.
Konstruktor NEMA povratnog tipa!
Konstruktor MORA biti javna funkcija!
Konstruktor MORA imati isto ime kao klasa!

Moguće je da neka funkcija ima više konstruktora, ali tad oni moraju primati različiti broj ili tip parametara (dovoljan je i samo različit redoslijed tipova argumenata) (ovo je polimorfizam, također jedna od važnih mogućnosti OOP-a), npr.:

Kod:

```
class Konstruktori {
        private:
                int x;
                int y;
                int n;
                char c;
                float f;
        public:
                // legalne definicije konstruktora
                Konstruktori() {
                        this->y = 444;
                        this->c = 'g';
                }
                Konstruktori(int x) {
                        this->x = x;
                        this->n = x;
                        this->y = 100;
                }
                Konstruktori(float f) {
                        this->f = f;
                }
                Konstruktori(int x, int y) {
                        this->x = x;
                        this->y = y;
                }
                Konstruktori(int x, char c) {
                        this->x = x + 5;
                        this->c = c;
                }
                /*      Primijeti raspored tipova argumenata
                        gornjeg i ovog konstruktora.
                */
                Konstruktori(char c, int x) {
                        this->c = c;
                        this->x = x*2;
                }

                // Ilegalni konstruktori (ukoliko smo već
                // definirali gornje):
                /*      Imamo već definiran konstruktor koji prima
                        argument (int x), a nas zanima samo tip
                        argumenta, a ne njegov naziv
                */
                Konstruktori(int y) {
                        this->y = y;
                }

                /*      Imamo već definiran konstruktor koji prima
                        argumente (int x, int y), a to promatramo kao:
                        (int, int) (sjetite se iz PIPI-a kako smo mogli
                        definirati prototipove funkcija bez naziva
                        varijabli).
                */
                Konstruktori(int y, int x) {
                        this->y = y;
                        this->x = x;
                }
};
```

Copy konstruktor:
Kako su objekti zatvorene cjeline, i kako se u njih može osim primitivnih tipova (int, char, int*, float*, double, ...) mogu spremiti i složeni tipovi podataka (npr. razna polja) dolazi do problema pri njihovom fizičkom kopiranju (točnije do (obično) ne željenih rezultata).
Npr.:
Kod:

```
class CopyKons {
        private:
                int *polje;
                int brojElemenata;
                int x;
        public:
                CopyKons() {
                        this->brojElemenata = 3;
                        this->polje = (int *) malloc
                                        (brojElemenata*sizeof(int));
                        *(this->polje + 0) = 4;
                        *(this->polje + 1) = 5;
                        *(this->polje + 2) = 6;
                        this->x = 1;
                }
                void setX(int x) {
                        this->x = x;
                }
                // funkcija na mjesto polje[1] stavlja "el"
                void zamjeniEl(int el) {
                        *(this->polje + 1) = el;
                }
                void ispis() {
                        cout << this->x << "  "  << *(this->polje + 1);
                }
};

void fPrimitivna(int x) {
        x = 500;
}

void fObjektna(CopyKons obj) {
        obj.setX(500);
        obj.zamjeniEl(500);
}

int main() {
        CopyKons obj;
        int x = 10;

        fPrimitivna(x);
        cout << x << endl;
        // ispisuje: 10

        fObjektna(obj);
        obj.ispis();
        // ispisuje: 1, 500
        // funkcija nije globalno promjenila
        // x, ali jest 2. član polja!

        return 0;
}
```

Evo o čemu se radi. Kad pozovete neku funkciju sa argumentima, vrijednosti poslanih argumenata se kopiraju i spremaju fizički u memoriju kao nove varijable, te ih je moguće mijenjati samo na lokalnoj razini (za pokazivače bi ovo značilo mijenjanje same adrese!). Ali, budući da mi šaljemo objekt, tj. složenu strukturu koja sadrži i primitivne i složene tipove podataka, stvara se novi objekt koji sadrži jednake vrijednosti varijabli PRIMITIVNIH tipova, a time i adresu unutar pokazivača! Tako da sad imamo 2 pokazivača koja pokazuju na istu memoriju, ali se nalaze u različitim objektima, i mijenjanjem jednog od njih memorija se globalno mijenja.
Ovo obično nije poželjno, zato se koristi copy konstruktor koji se poziva kad se kao argument funkciji proslijedi objekt. On služi za "reinicijalizaciju" te kopije objekta. Njime pazimo da nam naš objekt i dalje ostane zatvorena cjelina.
Primjer kad su nam copy konstruktori potrebni je klasa koja sadrži polje. Sad bi se pozivanjem copy konstruktora trebala rezervirati nova memorija za polje i kopirati svi elementi starog polja u novo. Znači, gornja klasa bi trebala imati i konstruktor:
Kod:

```
CopyKons(const CopyKons& obj) {
        int i;
        // primijeti gdje se u ovoj liniji koristi obj!
        this->polje = (int *) malloc (obj.brojElemenata * sizeof(int));

        for (i = 0; i < obj.brojElemenata; i++) {
                *(this->polje + i) = *(obj.polje + i);
        }
}
```

Nakon ovoga funkcija fObjektna(CopyKons obj) ne bi promijenila globalnu vrijednost 2. člana polja, već bi ta promjena ostala na lokalnoj razini funkcije (budući da smo sad napravili pravu (deep) kopiju objekta).

Destruktori:
Oni su funkcije koje se izvršavaju pred uništavanje objekta (o "životu" objekta više u sljedećem dijelu).
Obično služe za oslobađanje memorije (ukoliko smo unutar objekta posebno zauzimali memoriju) i zatvaranje vanjskih resursa koje je objekt otvorio (npr. neka datoteka).
Ako vam objekt ne koristi neke vanjske resurse, a svi podatci mu se nalaze samo u obliku primitivnih tipova, destruktor vam ne treba (tj. tad koristite defaultni, odnosno praznu funkciju). Ali, pazite! Ako koristite polja konstantne veličine, njih također NE trebate (niti možete) "osloboditi" ručno.
Destruktor za gornji primjer bi izgledao:
Kod:

```
// primjeti znak "~" ispred imena.
// CopyKons je ime klase!
~CopyKons() {
        free(this->polje);
}
```

Što se tiče imena, modifikatora pristupa i povratne vrijednosti, za destruktor vrijede ista pravila kao i za konstruktor.

3. Stvaranje i uništavanje objekata

Stvaranje objekta znači zauzimanje memorije za njegove varijable, te pozivanje nekog konstruktora radi inicijalizacije.
Pazite, objekt ne možete stvoriti preko malloc-a, jer malloc samo rezervira memoriju za resurse, ali ne poziva konstruktor, a konstruktor se ne može direktno pozvati.
Objekt možete stvoriti na dva načina, jedan sprema resurse na stog, drugi na heap ("hrpu"). I jedan i drugi način stvaranja će prvo zauzeti memoriju, a nakon toga pozvati konstruktor.
Za razumijevanje stoga i hrpe pogledajte slide-ove (ASP-3-PoziviFunkcije-Stack, slide 5.).
Ako se objekt stvori na stogu u main funkciji, on će se uništiti (pozvati destruktor, a nakon toga dealocirati resursi (ilitiga osloboditi memorija)) nakon što main funkcija završi sa radom (nakon "return").
Objekte stvorene na heap-u moramo sami brisat pozivanjem operatora "delete". Ukoliko ih ne obrišemo sa "delete"-om, destruktor im neće biti pozvan.

Primjer stvaranja objekta na stogu:
Kod:

```
class Razred {
        public:
                Razred() {
                        cout << "def" << endl;
                }
                Razred(int x, int y) {
                        cout << "2 inta" << endl;
                }
                Razred(char c) {
                        cout << "char" << endl;
                }
                void ispisi(int x) {
                        cout << x << endl;
                }
};
int main() {
        //      Stvaramo 3 objekta na STOGU
        //      2 preko default konstruktora,
        //      jedan preko konstruktora koji prima 1 char
        //      PAZI! Ako se poziva defaultni konstruktor
        //      zagrade se NE smiju pisati!
        Razred r1, r2, r3('a');
        //      ovo će ispisati (pogledaj kod konstruktora):
        //      def
        //      def
        //      char

        //      Objektima r1, r2 i r3 će se pozvati
        //      destruktor i dealocirati memorija
        //      nakon što se izvrši naredba
        //      return 0;

        // Stvaramo niz objekata.
        // Defaultni konstruktor se poziva za svaki objekt.
        Razred rNiz[2];

        // Primjer pristupanja članskoj funkciji:
        rNiz[1].ispisi(2);
        return 0;
}
```

Primjer stvaranja objekta na heap-u:
Kod:

```
class Razred {
        public:
                Razred() {
                        cout << "def" << endl;
                }
                Razred(int x, int y) {
                        cout << "2 inta" << endl;
                }
                Razred(char c) {
                        cout << "char" << endl;
                }
                void ispisi(int x) {
                        cout << x << endl;
                }
};

int main() {
        /*      Stvaramo objekt na HEAP-u
                PAZI! Ako se poziva defaultni konstruktor
                zagrade se MORAJU pisati!
                Primijeti da je sad r1 pokazivač.
        */
        Razred *r1 = new Razred();
        // ispisuje: def

        Razred *r2 = new Razred('c');
        // ispisuje: char

        /*      Objektima *r1 i *r2 će se pozvati
                destruktor i dealocirati memorija
                nakon što se izvrši naredba
                delete r1;
                odnosno
                delete r2;
        */
        // Stvaramo niz objekata.
        // Defaultni konstruktor se poziva za svaki objekt.
        Razred *rNiz = new Razred[2];
        // Objekti niza rNiz će se uništiti nakon
        // poziva operacije: delete [] rNiz;

        // Primjer pristupanja članskoj funkciji:
        rNiz[1].ispisi(5);

        // ALI! Ako nemamo niz objekata:
        r1->ispisi(10);
        // ili, alternativno:
        (*r1).ispisi(10);

        // Brisanje objekata
        // Primijetite da se šalje ADRESA
        delete r1;
        delete r2;
        delete [] rNiz;

        return 0;
}
```

4. Getteri i setteri:

Oni su osnovne funkcije javnog sučelja objekta.
Preko njih KONTROLIRANO postavljamo ili očitavamo vrijednosti privatnih varijabli objekta.
Time se brinemo da nam je objekt zaštićena cjelina sa smislenim podatcima, tj. ako imamo neku klasu Student, i ona uz ime i prezime sadrži JMBAG, a znamo da je JMBAG 10-eroznamenkati broj, vi morate spriječiti svaki pokušaj neispravnog unosa. Na taj način olakšavate sebi posao, jer ako znate da vam objekt sadrži (bar formatom) ispravne podatke vi te iste više ne morate provjeravati prije korištenja.
Evo koda:
Kod:

```
// recimo da je student definiran samo JMBAG-om
// i brojem bodova
class Student {
        private:
                float brBodova;
                // budući da JMBAG uvijek zauzima 10 znakova,
                // možemo to napraviti konstantnim nizom
                char JMBAG[10];
        public:
                // budući da želim prikazati gettere
                // i settere, konstruktor preskačem
                /*      Setter broja bodova.
                        setImeVarijable() je konvencija (dogovor).
                        Funkcija postavlja brBodova na broj
                        predat argumentom.
                */
                void setBrBodova(float brBodova) {
                        this->brBodova = brBodova;
                }
                /*      Setter JMBAG-a studenta.
                        Ukoliko postavljanje uspije
                        vraća 1, inače 0.
                        (ovo se trebalo riješiti iznimkama (isto jedna
                        korisna stvar), ali zasad nam nije to potrebno
                        i ne želim vas zamarati time).
                */
                int setJMBAG(char *JMBAG) {
                        int i;

                        if (strlen(JMBAG) != 10) return 0;
                        for (i = 0; i < 10; i++) {
                                if (!(*(JMBAG + i) >= '0' &&
                                                *(JMBAG + i) <= '9')) {
                                        return 0;
                                }
                        }
                   /*  JMBAG je prošao sve provjere. Postavimo ga.
                       PAZI! U ovom slučaju memorija je već zauzeta
                       jer koristimo konstantni niz znakova, ali
                       kad bi željeli imati proizvoljno dug niz,
                       trebali bi prvo alocirati memoriju (a poslije je i
                       u destruktoru osloboditi).
                    */
                        strcpy(this->JMBAG, JMBAG);
                        return 1;
                }
```

```
                // Getter broja bodova.
                float getBrBodova() {
                        return this->brBodova;
                }
};

int main() {
        Student s1;

        s1.setBrBodova(2.3f);

        // Ispisuje "Greska!"
        if (s1.setJMBAG("912jd"))
                cout << "Postavio sam JMBAG" << endl;
        else
                cout << "Greska!" << endl;

        // Ispisuje "Postavio sam JMBAG"
        if (s1.setJMBAG("0036123456"))
                cout << "Postavio sam JMBAG" << endl;
        else
                cout << "Greška!" << endl;

        // Ispisuje 2.3
        cout << s1.getBrBodova();

        return 0;
}
```

To bi bilo uglavnom sve...
Nadam se da je shvatljivo. Da bi napisao išta bolje trebala bi mi još bar 2-3 dana... jer, OOP je jednostavan za rad, ali nije lagan za objasniti...

Sretno svima na međuispitima!
Pozdrav,
Ivan