Elektrotehnički fakultet Osijek
Kneza Trpimira 2b
31000 Osijek

www.etfos.hr

1. UVOD U C++

1.1 Strukturno programiranje i C++

U programiranju danas dominiraju dva osnovna koncepta : koncept strukturnog programiranja i koncept objektno-orijentiranog programiranja. Pritom objektno orijentirano programiranje predstavlja nadgradnju koncepta strukturnog programiranja, obogaćujući ga novim mogućnostima i novim pristupom programiranju.

Tri osnovna koncepta strukturnog programiranja su :

- stroga hijerarhijska struktura programa (što znači da je upotreba naredbe GO TO nepoželjna, ili barem svedena na minimum),
- odvajanje definicije podataka od njihove obrade (odnosno, postoje odjeljci u programu za definiciju podataka i odjeljci za njihovu obradu) i
- potprogrami. Logičke cjeline unutar programa izdvajaju se u potprograme s točno određenom zadaćom, tako da se isti programski kod može pozivati iz različitih dijelova programa.

1.2 Odnos između C i C++

Kao što je C strukturno-orijentirani programski jezik, tako je C++ (uz Javu, koja, međutim, nasljeđuje većinu koncepata i sintaksu od C++) najtipičniji objektno-orijentirani programski jezik. C++ predstavlja proširenje programskog jezika C, u kojeg uvodi koncept objektno-orijentiranog programiranja. Jezik C ostao je i dalje uključen u jezik C++, kao njegov podskup. Kao takav, C++ podržava oba programska koncepta : koncept strukturnog programiranja i koncept objektno-orijentiranog programiranja.

1.2.1 Komentari

Osim oznaka /* i */ iz C-a koje predstavljaju početak, odnosno kraj bloka koji predstavlja komentar, C++ uvodi i znak // koji označava red izvornog koda programa kao komentar. Na primjer :

```
{
// Ovo je komentar!
}
```

1.2.2 Dodjela tipa podataka

U C-u se dodjela tipa vrši tako da se tip navede u zagradi ispred naziva varijable, kao u sljedećem primjeru :

```
int i; float f;
f = (float) i;
```

C++ dopušta i drugi način, po uzoru na poziv funkcije :

```
int i; float f;
f = float (i);
```

1.2.3 Ulaz i izlaz

Ulaz i izlaz je u C++ riješen pomoću ulaznih i izlaznih tokova. Tokovi su klase definirane u standardnim bibliotekama, koje omogućuju komunikaciju programa s vanjskim jedinicama (npr. tipkovnicom i ekranom).

Ispis pomoću cout

Jedna od najočitijih razlika između C i C++ je zamjena standardne biblioteke *stdio* bibliotekom *iostream*. Biblioteka *iostream* zamjenjuje sve mogućnosti biblioteke *stdio*, odnosno, tri programska retka iz slijedećeg primjera daju identičan rezultat :

```
printf ("Danas je lijep dan.\n");          // C
cout << "Danas je lijep dan.\n";           // kombinirano rješenje
cout << "Danas je lijep dan." << endl;     // C++
```

Znakovi “<<” predstavljaju operator umetanja u tok, jer se njime podatak umeće u tok. S desne strane se može naći bilo koji tip podatka: char, short, int, long int, char *, float, double, long double, void * . Tako vrijede sljedeće operacije:

```
cout<<9.3<<endl;                                  //float
cout<<615 <<endl;                                 //int
cout<<"Ovo je ispis znakovnih nizova"<<endl;      //char *
```

Za formatiranje izlaza mogu se koristiti manipulatori, koji su definirani unutar biblioteke *iomanip*.

Primjer :

```
#include <iostream>
#include <iomanip>
using namespace std;

int main () {
  double pi=3.141548;
  cout << setprecision (5) << pi << endl;
  cout << setprecision (9) << pi<< endl;
```

```
  cout << fixed;
  cout << setprecision (5) << pi<< endl;
  cout << setprecision (9) << pi << endl;
  return 0;
}
```

Gornji primjer ispisati će sljedeće:

3.1415
3.141548
3.14155
3.141548000

Postoji više manipulatora: setw(int), dec, hex, oct, flush, endl...

Učitavanje pomoću cin

Ulazom se upravlja na sličan način kao s ispisom, s tim da se koristi ulazni tok *cin* i operator izlučivanja iz toka (>>) jer se njime podatak izlučuje s izlaznog toka.

Primjer :

```
scanf ("%f",&pi);      // C
cin >> pi;             // C++
```

S desne strane se može naći bilo koji tip podatka: char, short, int, long int, char *, float, double, long double, void * . Tako vrijede sljedeće operacije:

```
int i;
long l;
float f;
double d;

cin>>i;          //int
cin>>l;          //long int
cin>>f;          //float
cin>>d;          //double
```

Uspješnim učitavanjem podatka operator izlučivanja kako rezultat vraća referencu na objekt tipa istream. Zbog toga se upis može ulančavati pa možemo napisati:

```
cin>>i>>l>>f>>d;
```

Datotečni ulaz/izlaz

Umjesto biblioteke *stdio* koristi se biblioteka *fstream* (također se mogu koristiti *ifstream* i *ofstream*). Sljedeći primjer u C-u upisuje dva reda teksta u izlaznu tekstualnu datoteku :

```
FILE *dat;
dat = fopen ("tekst1.txt","w");
fprintf (dat,"%s","Prvi red teksta!\n");
fprintf (dat,"%s","Drugi red teksta!\n");
fclose(dat);
```

Odgovarajući primjer u C++ :

```
fstream dat;
dat.open ("tekst.txt",ios::out);
dat << "Prvi red teksta!" << endl;
dat << "Drugi red teksta!" << endl;
dat.close();
```

U drugom primjeru definiran je datotečni objekt *dat*, iz klase *fstream.* Za otvaranje datoteke korišten je funkcijski član *open*. Datoteka je otvorena u modu *out* (izlazna datoteka). Tekst se u datoteku upisuje korištenjem operatora umetanja u tok (<<). Da smo željeli čitati iz datoteke koristili bi mod *ios::in*.

Deklaracije varijabli

Varijable se u C++ deklariraju na isti način kao i u C-u, ali mogu biti deklarirane u bilo kojem dijelu programskog koda.

Primjer:

```
{
int a;
... programski kod ...
float b;
... programski kod ...
char c;
... programski kod ...
}
```

Konstante

U C-u se za definiranje konstanti koristi pretprocesorska naredba *#define*. Na primjer:

```
#define UKUPNO 50
```

U tu svrhu u C++ se može koristiti i ključna riječ *const* :

```
const UKUPNO = 50;
```

Također, C++ dopušta da pojedini argumenti funkcija budu konstantni, time se sprječava promjena njihove vrijednosti unutar funkcije. Primjer :

```
void funkcija (const int a)
{
a=10;          // Greška; ne može se promijeniti konstanta
}
```

Preopterećenje funkcije

C++ omogućuje da više funkcija ima isti naziv, pod uvjetom da su im liste parametara različite. Prema listi parametara kod poziva funkcije određuje se koja će funkcija biti pozvana.

Primjer :

```
#include <iostream>
using namespace std;

void funkcija(){
cout << "Funkcija bez parametara!" << endl;
}
void funkcija(int a){
cout << "Cjelobrojni parametar a = " << a << endl;
}
void funkcija(float b, float c){
cout << "Realni parametar b = " << b << endl;
cout << "Realni parametar c = " << c << endl;
}
void main() {
funkcija();
funkcija(10);
funkcija(2.71,3.14);
}
```

Ispisuje se :

Funkcija bez parametara!
Cjelobrojni parametar a = 10
Realni parametar b = 2.71
Realni parametar c = 3.14

Podrazumijevani argumenti funkcija

Prilikom poziva funkcije potrebno je navesti listu argumenata koja odgovara listi argumenata u zaglavlju funkcije. C++ dopušta da se neki od argumenata u pozivu funkcije izostave, tako da se umjesto njih koriste podrazumijevane vrijednosti.

Primjer:

```
#include <iostream>
using namespace std;
void funkcija(int a, int b=5){
cout << "a = " << a << endl;
cout << "b = " << b << endl << endl;
}
void main(){
funkcija (3,4);
funkcija (10);
}
```

Ispisuje se :
a = 3
b = 4
a = 10
b = 5

U drugom pozivu funkcije izostavljen je drugi argument, umjesto kojeg se koristi podrazumijevani argument (b=5).

Alokacija memorije

C++ zamjenjuje C-ovu funkciju za alokaciju memorije *malloc* i funkciju za dealokaciju memorije *free* s ***new*** i ***delete***. New i delete omogućuju alokaciju korisnički definiranih tipova jednako kao i preddefiniranih.

C:

```
int *pok;
pok = (int*)malloc(sizeof(int));
*pok = 10;
free (pok);
```

C++ :

```
int *pok;
pok = new int;
*pok = 10;
delete pok;
```

Deklaracije referenci

U C-u se često koriste pokazivači za prosljeđivanje argumenata funkcijama. U C++ može se koristiti referentni operator (&) u listi argumenata, što čini programski kod čišćim.

C:

```
void zamjena (int *a, int *b){
int t = *a;
*a = *b;
*b = t;
}
void main(){
int x = 3, y = 5;
zamjena (&x, &y);
printf ("%i %i\n",x,y);
}
```

C++:

```
void zamjena (int &a, int &b){
int t = a;
a = b;
b = t;
}
void main(){
int x = 3, y = 5;
zamjena (x,y);
cout << x << y << endl;;
}
```

Također, moguće je referenci varijable pridružiti varijablu, kao u slijedećem primjeru :

```
int a = 0;
int &b = a;     // varijable a i b zauzimaju isti memorijski prostor
b = 5;
cout << a << endl;    // ispisuje se 5
```

2. OBJEKTI I KLASE

2.1. Uvod u objektno programiranje

Objektno orijentirano programiranje: Postupak izrade aplikacija u kojem se svojstva apstraktnih ili stvarnih objekata modeliraju programskim klasama i objektima. U stvarnom svijetu okruženi smo objektima (automobil, računalo, pas, drvo, ...). Svaki objekt definiran je stanjem i ponašanjem. Npr. za automobil stanje određuje trenutna brzina, količina goriva u spremniku i sl., a ponašanje može biti ubrzavanje, kočenje, skretanje i sl.

Objekt u programskom okruženju je skup varijabli i pripadnih metoda. Varijable određuju stanje, a metode ponašanje objekta. Varijable objekta nazivaju se i varijable primjerka (*instance variables*) – svaki primjerak (instanca) određenog objekta sadrži vlastitu kopiju varijabli primjerka. Metode mijenjaju stanje objekta, a po potrebi mogu stvarati i nove objekte.

Objekti međusobno razmjenjuju informacije i traže jedan od drugog usluge. Pritom okolina ne mora znati ništa o unutanjem ustrojstvu objekta. Okolina komunicira sa objektom preko *javnog sučelja* (engl. public interface). Način na koji se ostvaruje reprezentacija objekta jest *implementacija objekta* (engl. implementation) i ona se najčešće skriva od okoline da bi se osigurala konzistentnost objekta. Klasa se dakle sastoji od opisa javnog sučelja i od implementacije. Objedinjavanje sučelja i implementacije naziva se enkapsulacija (engl. encapsulation).

Klasa je nacrt objekta, odnosno predložak po kojem je određeni objekt stvoren. Npr. određeni tip automobila definiran je jednim nacrtom (odnosno skupom nacrta). Na temelju jednog nacrta moguće je proizvesti više primjeraka istog tipa automobila (tj. objekata), koji će se međusobno razlikovati po stanju i ponašanju. Podskup stanja i ponašanja (varijabli i metoda) može biti zajednički svim objektima određene klase. Nazivamo ih varijablama i metodama klase.

Sintaksa za kreiranje klase u C++ je vrlo slična sintaksi za kreiranje strukture u C-u:

```
Struct datum
        {
                int dan, mjesec, godina;
        };
```

Ali pri kreiranju klase osim varijabli definiramo i metode kojima se objekt služi kako bi pristupio svojim varijablama i mijenjao ih.

```
class datum
        {
                int dan;        // Varijable koje se koriste za pohranu datuma
                int mjesec;
                int godina;

public:         // Metode klase
                datum();                                    //constructor
                datum(int,int,int);
                void postavidatum (int, int, int);          //prototip funkcije
                void pokazi();                              //prototip funkcije
```

Kada definiramo klasu možemo definirati i njene metode. Definiranje metode ima sljedeći format:

```
Povratni tip ime klase::ime metode (parametri) {
        Tijelo metode
}
```

Slijedi definicija metoda za naš primjer:

```
void datum::postavidatum (int dd, int mm, int yy)
{
        dan=dd;
        mjesec=mm;
        godina=yy;
}

void datum::pokazi (void)
{
cout<<"Dan:"<<dan<<"\n"<<"Mjesec:"<<mjesec<<"\n"<<"Godina:"<<godina;
}
```

2.2. Konstruktori

Konstruktor je posebna metoda, čije ime je jednako imenu klase, koja se automatski izvršava u trenutku stvaranja objekta. Najčešće se koristi za inicijalizaciju varijabli objekta. U pravilu se definira više konstruktora.

```
datum::datum ()
{
        dan=1;
        mjesec=1;
        godina=2008;
}
```

U gornjem primjeru konstruktor inicijalizira datum na 1.1.2008. godine.

- Svaka klasa mora imati najmanje jedan konstruktor.
- Ime konstruktora mora biti jednako imenu klase.
- Iako nije prikazano u gornjem primjeru konstruktor može primati i parametre.
- Konstruktor ne vraća nikakvu vrijednost (čak niti void).
- Objekt se može stvoriti samo jednom kao što se i varijabla samo jednom može definirati.

2.3. Stvaranje objekta

Nakon što smo definirali klasu možemo stvoriti pojedinačne objekte tako da navedemo ime klase kojoj objekt pripada i ime objekta:

datum danas;

- Kreirali smo objekt klase **datum** koji se zove **danas**
- Budući da je objekt **danas** instanca klase **datum** pri njegovoj se deklaraciji automatski poziva konstruktor koji postavlja vrijednost datuma na 1.1.2008.

- Objekt **danas** izgleda ovako:

danas
dan=1 mjesec=1 godina=2008
void postavidatum(int,int,int) void pokazi()

- Kao i kod struktura možemo pristupati pojedinačnim varijablama objekta koristeći sljedeću notaciju "."

danas.dan=15;

- No rijetko ćemo direktno pristupati varijablama objekta jer tako ne koristimo prednost objektno orijentiranog programiranja. Umjesto toga ćemo pozvati metodu objekta. Sintaksa poziva metode je:

Ime objekta.ime metode (parametri);

danas.postavidatum(15,10,2008);

- Objekt **danas** sada izgleda ovako:

danas
dan=15 mjesec=10 godina=2008
void postavidatum(int,int,int) void pokazi()

- Za ispis trenutnog datuma:

danas.pokazi();

2.4. Preopterećenje konstruktora

U C++ moguće je preopterećenje konstruktora. Da bismo to pokazali dodat ćemo još jedan konstruktor postojećoj klasi.

```
//Stari konstruktor koji postavlja datum na predefiniranu vrijednost
datum();                        //ovo je samo prototip konstruktora

//Novi konstruktor koji stvara objekt datum sa definiranim vrijednostima za dan,
mjesec i godinu

datum(int dd, int mm, int yy);      //ovo je samo prototip konstruktora

datum::datum(int dd, int mm, int yy)
{
        dan=dd;
        mjesec=mm;
        godina=yy;                  //ovo je definicija konstruktora
}
```

Primjer kako pozvati oba konstruktora:

```
datum petak;                //ovo će pozvati konstruktor sa predefiniranim
                            //vrijednostima
datum danas(15,11,2007);    //ovo će pozvati preopterećeni konstruktor
```

2.5. Destruktor

Kao što se objekt može stvoriti tako se mora moći i uništiti. C++ omogućuje eksplicitno korištenje destruktora, tj. metode koja se poziva neposredno prije uništenja objekta.

Sintaksa za destruktor je vrlo jednostavna:

```
~ime klase();
```

U našem primjeru:

```
~datum();
```

Destruktori se koriste za čišćenje memorije nakon što su objekti uništeni.

2.6. Dodjela prava pristupa

Apstrakcija je mogućnost uklapanja tuđeg programa u svoj vlastiti program bez razumijevanja kako je program implementiran. Moramo znati što program radi, ali ne moramo znati kako on radi. Ljudi koji rade velike programe moraju brinuti samo kako radi njihov kod i kako koristiti kod koji su napisali drugi ljudi, a ne moraju razumjeti čitav program.

Ovijanje (engl. encapsulation) je mehanizam kontrole pristupa varijablama i metodama klasa i objekata. Ona odvaja implementaciju od korisnika. Svaki pristup unutarnjoj reprezentaciji je kroz metode klase koje predstavljaju sučelje prema korisniku. Dva su razloga zbog kojih korisnik ne bi trebao imati pristup unutarnjoj prezentaciji objekta:

- Ovijanje sprječava korisnika da mijenja podatke na ilegalan način
- Ovijanje omogućava programeru da mijenja stanje objekta u bilo koje vrijeme

C++ implementira enkapsulaciju korištenjem ključnih riječi **public**, **private** i **protected**. Prava pristupa određuju koji će članovi razreda ili klase biti dostupni izvan razreda, koji iz naslijeđenih razreda, a koji samo unutar razreda. Svaki član klase može imati jedan od tri moguća načina pristupa što će biti pokazano na primjeru:

Primjer:

```
class Pristup {
public:
        int a, b;
        void Funkcija1(int brojac);

private:
        int c;

protected:
        int d;
        int Funkcija2();
};
```

- Javni pristup se dodjeljuje ključnom riječi **public**. Članovima sa javnim pristupom može se pristupiti i izvan klase. Varijable a i b te Funkcija1() se mogu pozvati tako da se definira objekt klase **Pristup** i da im se pomoću operatora **.** pristupi.

- Privatni pristup se dodjeljuje ključnom riječi **private**. Članovi sa privatnim pristupom nisu dostupni vanjskom programu i klasama koji nasljeđuju promatranu klasu. Tako se varijabli c može pristupiti samo preko funkcijskih članova klase pristup.

- Zaštićeni pristup se dodjeljuje ključnom riječi **protected**. Članovi sa zaštićenim pristupom nisu dostupni vanjskom programu nego samo funkcijskim članovima klase i klasama koji nasljeđuju promatranu klasu.

Ako se eksplicitno ne navede pravo pristupa nekom članu klase podrazumijeva se privatno pravo pristupa. Za ilustraciju korištenja prava pristupa stvoriti ćemo objekt klase **Pristup** i pokazati kojim se članovima može pristupiti iz kojeg dijela programa:

```
//stvaranje objekta x koji pripada klasi pristup
Pristup x;
void Pristup::Funkcija1(int brojac) {
//sada smo unutar funkcijskog člana klase pa možemo
//pristupiti svim članovima klase
a=brojac;
c=a+5;
Funkcija2();
}
int main() {
x.a=x.b=6;              //U redu jer a i b imaju javni pristup
x.Funkcija1(1);         // U redu jer Funkcija1() ima javni pristup
cout<<x.c<<endl         //Nije u redu jer c ima privatni pristup i ne može
                        //mu se pristupiti izvana
x.Funkcija2();          //Nije u redu jer Funkcija2() ima zaštićeni pristup
};
```

2.7. Javno sučelje

Članovi razreda sa javnim pristupom formiraju javno sučelje objekta. Programer analizira objekte i način njihovog korištenja te tako određuje koji članovi pripadaju javnom sučelju. On zatim obznanjuje suradnicima što točno trebaju pružiti objektu i što od njega mogu dobiti nazad. Sam sadržaj objekta je crna kutija za korisnike. Implementaciju klase korisnik piše na osnovi javnog sučelja čime sučelje postaje neovisno o implementaciji. Ako se kasnije ustvrdi da je neka implementacija bolja objekt se jednostavno preradi, dok ostatak koda (npr. Kod drugih programera) ne treba dirati jer on vidi samo javno sučelje objekta koje se ne mijenja.

Primjer:

```
class Vektor {
private:
        float ax, ay;
public:
        void PostaviXY(float x, float y) {
                ax=x;
                ay=y;
        }
        float VratiX() { return ax; }
        float VratiY() { return ay; }
        void MnoziSkalarom(float skalar) {
                ax*=skalar;
                ay*=skalar;
        }
};
```

Implementacija vektora je izvedena u Descartesovom koordinatnom sustavu. Javno sučelje ne govori ništa o koordinatama nego samo omogućava da se utvrdi projekcija svakog vektora na os x i os y neovisno u kojem je koordinatnom sustavu vektor prikazan. Mi u implementaciji možemo promijeniti koordinatni sustav u polarni ako se pokaže da program bolje radi, ali javno sučelje ostaje isto.

Primjer:

```
#include <iostream>
using namespace std;
class Point {
        long double  x;
        long double  y;
public:
        Point();
        Point(int, int);
        ~Point();
};
Point::Point(){                                         //Definiranje konstruktora
        x=0;
        y=0;
        cout<<"Poziv konstruktora bez parametara"<<endl;
        cout<<"X="<<x<<"\tY="<<y<<endl;
}
Point::Point(int InitX,int InitY){                      //Preopterećeni konstruktor
        x=InitX;
        y=InitY;
        cout<<"Poziv preopterećenog konstruktora "<<endl;
        cout<<"X="<<x<<"\tY="<<y<<endl;
}
Point::~Point(){                                        //Definiranje destruktora
        cout<<"Poziv destruktora"<<endl;
}
void main(){
        Point P1;                               //Stvaranje objekta P1 prvim konstruktorom
        Point P2(10,10);                        //Stvaranje P2 preopterecenim konstruktorom
}
```

Zadatak:

a) Napraviti metodu koja će postavljati koordinate točke na zadanu vrijednost i koja će ispisati koordinate točke, tj. treba prepraviti gornju klasu (Promjene su prikazane crvenom bojom).

b) Dodati metodu koja će računati udaljenost izmedu dvije točke i ispisati udaljenost na ekran. Funkciju preopteretiti da računa udaljenost zadane točke od ishodišta koordinatnog sustava (Promjene su prikazane plavom bojom).

```
#include <iostream.h>
#include <math.h>
using namespace std;
class Point {
        long double  x;
        long double  y;
        long double d;
public:
        Point();
        Point(int, int);
        ~Point();
        void SetXY(int,int);                          //Deklaracija metode SetXY
        void Distance(Point,Point);
        void Distance(Point);
};

Point::Point(){                                       //Definiranje konstruktora
SetXY(0,0);
cout<<"Poziv konstruktora bez parametara"<<endl;
}

Point::Point(int InitX,int InitY){                    //Preopterećeni konstruktor
SetXY(InitX,InitY);                                   //Poziv metode SetXY
cout<<"Poziv preopterećenog konstruktora "<<endl;
}
void Point::SetXY(int Xset,int Yset) {                //Definiranje metode SetXY
x=Xset;
y=Yset;
cout<<"X="<<x<<"\tY="<<y<<endl;
}
void Point::Distance(Point P1, Point P2) {            //Definiranje metode Distance
d=sqrt((P1.x-P2.x)* (P1.x-P2.x) + (P1.y-P2.y)* (P1.y-P2.y));
cout<<"Udaljenost dvaju tocaka je d="<<d<<endl;
}
void Point::Distance(Point P1) {
d=sqrt((P1.x-0)* (P1.x-0) + (P1.y-0)* (P1.y-0));
cout<<"Udaljenost od ishodista je d="<<d<<endl;
}
Point::~Point() {                                     //Definiranje destruktora
cout<<"Poziv destruktora"<<endl;
}
void main() {
Point P1;                                   //Stvaranje objekta P1 prvim konstruktorom
Point P2(10,10);                            //Stvaranje P2 preopterećenim konstruktorom
Point P3(15,18);                            //Stvaranje P3 preopterećenim konstruktorom
P1.SetXY(5,5);
P1.Distance(P1,P2);
P3.Distance(P3);
}
```