C++ miniTutorial

Osnovni elementi i koncepti programskog jezika C++ uz primere

Sadržaj

I. Ugrađeni tipovi podataka
II. Doseg (scope)
III. Životni vek objekta
IV. Konverzija tipa (casting)
V. Struktura programa
VI. Klase i objekti
VII. Korisni linkovi

I: Ugrađeni tipovi podataka

- Znakovni tip (char)

```
char c1 = '0'; // znak '0'
char c2 = '\0' // terminating char 0, koristi se da
                  označi kraj niza znakova (C string)
```

- Logički tip (bool)

```
bool flag1 = false;
bool flag2 = 0; // flag1 == flag2
bool flag3 = -12; // flag3 != flag1, svaka nenulta
                     vrednost je true
```

- Celobrojni tip (int)

```
int i = -127; // signed se podrazumeva
unsigned ui = 127; // unsigned int je uvek pozitivan
short si = 127; // signed short int, manji opseg
long li = 123456UL; // signed long int, veći opseg
ui=i; // Nije greška već samo upozorenje pri kompajl.
```

- Decimalni tip (double, float)

```
float pi = 3.14; // smanjena preciznost
double d = 1.5e-4; // standardna preciznost
long double dl = 1.3; // najveća preciznost
d = dl; // Nije greška već Compile Warning
```

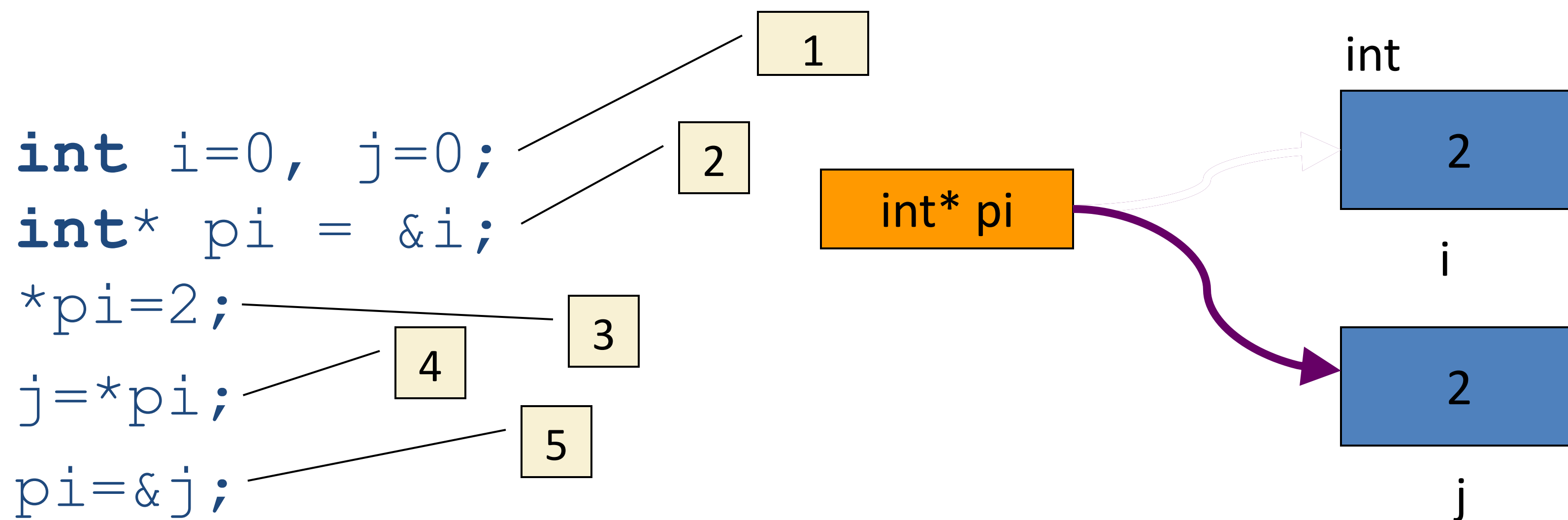
- Nabrajanja (enum)

```
enum Status { initiated, suspended,
            committed, canceled, failed };
// Enumeracije su niz dikretnih stanja
Status s1 = initiated; // s1 može imati vrednost samo iz
                          skupa Status
if (s1==committed) ...
Status s2 = canceled;
if (s2==4) ... // konverzija enum->int je OK
Status s3 = 3; // Error! (int->enum)
```

- Pokazivači (*)
  - Pokazivači su delovi memorije na kojima se čuvaju memorijske adrese podataka na koje pokazuju. Vrednost pokazivača (odnosno mem adresa) nam obično nije od interesa.

I-1: Pokazivači (*) i Upućivači (&)

- Ne postoji način da se proveri da li je pokazivač validan!
- Pokazivači su izvedeni tipovi podataka (ne postoji “pokazivač” već samo “pokazivač na neki tip podatka”)
- Dereferenciranje se vrši operatorom *

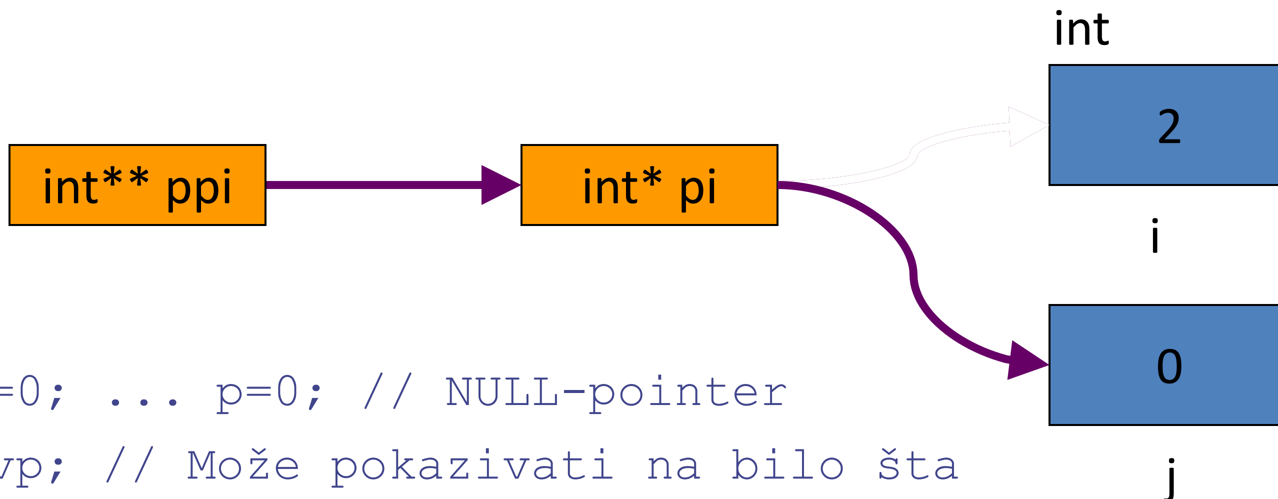


- Adresa podatka se dobija korišćenjem & operatora

```
&(*pi) == pi // Leva i desna strana su uvek jednake
```

- Pokazivači mogu pokazivati i na druge pokazivače:

```
int i=0, j=0;
int* pi=&i;
int** ppi; // "pokazivač na pokazivač na int";
ppi=&pi;
*pi=1;
**ppi=2; // Evaluated: *(*ppi)
*ppi=&j;
ppi=&i;   // Compilation error:
          // int* cannot be converted to int**
```




```
int* p=0; ... p=0; // NULL-pointer
void* vp; // Može pokazivati na bilo šta
             retko se koristi, ne postoje objekti tipa void
```

- Najčešće greške pri korišćenju pokazivača
  - Korišćennje neinicijalizovanog pokazivača

```
char* cp; // Nije inicijalizovan
*cp = 'M';  // Runtime error: Memory access
   violation ili pucanje programa(ne uvek)
```

  - Dereferenciranje NULL pokazivača

```
if (p!=0) ...p->... // Ovakve konstrukcije štite od
   Runtime error (najčešće hardware exception)!
```

  - Korišćenje “visećeg” pokazivača

```
vector<int>* vp1 = new vector<int> (5); // Alociraj
vector<int>* vp2 = vp1;
delete vp1; // Uništava objekat (pokazivači ostaju)
vp2->at(0); *vp2; // runtime error, pucanje, ili
                exception (ne uvek)
int* f() {
  int x = 5;
  return &x; // x se briše, jer ne postoji van f-je
}
...
int* p = f(); // p je viseći pokazivač!
```

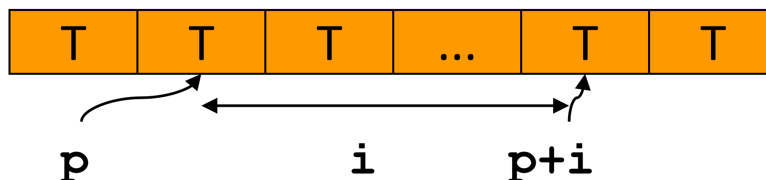

Marko Marković

I-2: Nizovi ([])

- Nizovi se sastoje iz uređene kolekcije objekata tačno definisane dužine (određene pri kreiranju)
- Nizovi su isto kao i pokazivači izvedeni tipovi podataka i ne treba ih mešati sa STL <vector>

```
int m[5][7]; // 5 x 7 matrix
m[3][5] = 2;
```

- Postoji jaka veza između nizova i pokazivača



  - (1) Kada se niz tipa T[] koristi on se implicitno konvertuje u pokazivač T* koji pokazuje na prvi element u tom nizu
  - (2) Za pokazivač tipa T* važi da ukoliko pokazuje na niz kretanje kroz isti je onda omogućeno jednostvnim uvećavanjem/umanjivanjem pok
  - (3) Izraz `a[i]` se shodno ovome tretira kao `*(a+i)` po definiciji

- Posledice povezanosti nizova i pokazivača:

```
int a[10];
int* p = &a; // p pokazuje na a[0]
a[2]=1;
p[3]=3;
p=p+1;
*(p+2)=1;
p[-1]=0;
```

Isto kao: = a; zato što se a automatski konvertuje u &a

- a[2]≡*(a+2) na osnovu #3
- u *(a+2), a se konvertuje u pokazivač na svoj prvi element tipa int* na osnovu #1
- pokazivaču se na osnovu #2 dodaje +2 i vrši se dereferenciranje

- p[3]≡*(p+3) na osnovu #3
- pokazivač se uvećava za 3 na osnovu #2, i rezultat se zatim dereferiše i dohvata se vrednost a[3]

Na osnovu #2, p sada pokazuje na sledeći element u nizu

Isto kao da smo naveli p[2]

Isto kao *(p-1).

- String je niz kara...

```
char* string = "Hello world"
```

- Celi nizovi se ne mogu prosleditni kao argumenti, već samo pokazivači (upućivači) na njih mogu

II: Oblast važenja (doseg)

- Globalna imena:
  - Imena koja se definišu van f-ja i klasa
  - Oblast važenja: od mesta deklaracije do kraja fajla
- Lokalna imena:
  - Imena deklarisana unutar bloka {} (npr. f-je)
  - Oblast važenja: od mesta deklarisanja do kraja bloka
- Sakrivanje imena:
  - Ako se redefiniše u unutrašnjem bloku, ime iz spoljašnjeg bloka je sakriveno do izlaska iz unutrašnjeg
- Pristup sakrivenom globalnom imenu:
  - Navođenjem :: operatora ispred imena

- Primer dosega i sakrivanja

```
int x  = 0;    // globalno x
void f() {
  int y = x,    // lokalno y, globalno x
      x = y;   // lokalno x, sakriva globalno x
  x = 1;       // pristup lokalnom x
  :: x = 5;    // pristup globalnom x
  {
       int x;   // lokalno x, sakriva prethodno lok. x
       x = 2;   // pristup drugom lokalnom x
  }
  x=3;         // pristup prvom lokalnom x
}
int *p = &x;    // uzimanje adrese globalnog x
```

II-1: Doseg klasa

- Oblast važenja klase imaju svi članovi klase
- Van tela klase promenljivima se pristupa pomoću
  - **.** , gde je levi operand objekat
  - **->** , gde je levi operand pokazivač na objekat
  - **::** , gde je levi operand ime klase

```
class X {
  int x;
  void f (); };
void X::f () { … x … } // :: prosirenje dosega
void g () {
  X xx, *px;
  px = &xx;
  xx.x = 0; // moze i xx.X::x, ali nema potrebe
  xx.f();   // moze i  xx.X::f() ali nema potrebe
  px -> x = 1; }
```

III: Životni vek objekata

- Po životnom veku objekti mogu biti
  - statički
    - je svaki globalni ili lokalni objekat deklarisan kao **static**
    - životni vek: od izvršavanja definicje do kraja programa!
    - globalni statički objekti se kreiraju pri kompajliranju dok se lokalni kreiraju pri prvom nailasku na njihovu definiciju
  - automatski
    - kreira se iznova pri svakom pozivu bloka u kome je deklarisan (npr. u for petlji, f-ji, ...) i automatski se briše
  - dinamički
    - životni vek kontroliše programer
    - kreiranje pomoću operatora **new** a destrukcija pomoću **delete**
  - privremeni
    - kreiraju se pri izračunavanju izraza i služe za smeštanje međurez.

- Primer automatskih i statičkih objekata

```
int a = 1;
void f() {
  int b = 1; // inicijalizuje se pri svakom
                pozivu - automatski
  static int c = 1; // inicijalizuje se samo
                       jednom
  cout << "a="<<a++<<" b="<<b++<<" c="<<c++<<endl
}
void main () {
  while (a<3) f();
}

izlaz:
a = 1 b = 1 c =1
a = 2 b = 1 c =2
```


Marko Marković

- Primer kreiranja i brisanja dinamičkih objekata

```
int* p1 = new int(3); // OK, jer new vraća
     // pokazivač na alociranu memoriju
int  p2 = new int(3); // compilation error:
                      // incompatible types
delete p1; // Ovako se brise jedan objekat
char* str2 = new char[20]; // alociranje 20 karaktera
str2 = strcpy(str2,"Marko"); // Dodela vrednosti
delete [] str2; // Ovako se brise niz objekata
```

- Dinamički objekti ostaju u memoriji i nakon što se napusti doseg f-je u kojoj su deklarisani jer ne postoji mehanizam implicitne destrukcije kao kod automatskih objekata

```
Counter* pc = 0;
  void f() {
    pc = new Counter(2);
  } // Dinamicki objekat je kreiran unutar f-je
  void main () {
    f();
    delete pc; // moramo obrisati pc
  }
```

IV: Konverzija tipa (kastovanje)

- Eksplicitne konverzije:
  - **(tip)** izraz // stari (C) način, ne preporučuje se
  - **static_cast** <oznaka_tipa> (izraz)
    - između numeričkih tipova
    - između pokazivača proizvoljnih tipova i void*
    - nestandardne konverzije (koje sami definišemo)
  - **const_cast** <oznaka_tipa> (izraz)
    - namenjen uklanjanju ili dodavanju const tipa
  - **dynamic_cast** <tip_pokazivaca_ili_ref> (izraz)
    - za podatke dinamičkog tipa (kreirane pomoću **new**)
- Impicitne konverzije su staičkog tipa i obavlja ih kompajler automatski (ukoliko su bezbedne)

- Primeri konverzije podataka

```
float a = 5.0; int i = static_cast<int> (a);
void* p = static_cast<void *>(&i);
// da smo naveli void* p = &i; cast bi bio impl
int* q = static_cast<int *> (p);
// morali smo eksplicitno

int j = 1; const int i = j;
int* p = const_cast<int *>(&i); // mora ekspl
*p = 0; // promena const podatka
double pi = 3.14;
const double &cpi = const_cast<const double &> (pi);
pi = 0.0; // OK
cpi = 0.0; // ! Error
```

V: Struktura programa

- (.c, .cpp, .cxx, .cc, .C) source - sadrži izvorni kod
- (.h, .hpp, .hxx) header - sadrži definicije funkcija i tipove podataka koji se nalaze u drugim datotekama
- Kreiranje programa se odvija u 2(3) faze:
  - **Preprocesiranje**
    - uzima source kod pretvara ga u text i prosleđuje ga kompajleru
    - preprocesor se rukovodi posebnim direktivama koje uvek počinju sa **#** i završavaju se nailaskom na kraj linije
  - **Kompajliranje**
    - prevodi kod u mašinski jezik i kreira (.obj) fajlove
    - opseg kompajlera je uvek svaki fajl ponaosob (ne razmatra se njihova međusobna zavisnost)
  - **Linkovanje**
    - objedinjuje .obj fajlove , razmatra njihovu međuzavsisnost, i kreira biblioteke (.so, .lib, .dll) i/ili izvršni program (.exe)

V-1: Preprocesor (#)

- Najčešće preprocesorske direktive:
  - **#include** - uključuje sadržaj fajlova u kod u kome je navedena direktiva. Ukoliko su direktive ugnježdene proces se ponavlja dok se ne stigne do poslednje
    - **#include** <filename>; // pretraga počinje od predefinisanih (sistemskih) foldera pa se proširuje na ostale (najčešće korisnički definisane)
    - **#include** "filename"; // obrnut smer pretrage
  - **#define** - definiše simbol ili vrši zamenu jednog simbola sa drugim (podržava parametrizaciju)
  - **#ifdef**/**#endif** i **#ifndef**/**#endif** - text unutar direktive se prosleđuje kompajleru samo ukoliko traženi simbol (ni)je definisan

- Primer korišćenja preprocerskih direktiva

```
#define _A_h
#define N 10
#define max(a,b) (((a)>=(b))?(a):(b))

int a[N];

void f (int a, int b){
   ...
   int c = max(x/3,y+1);
   ...
}
```

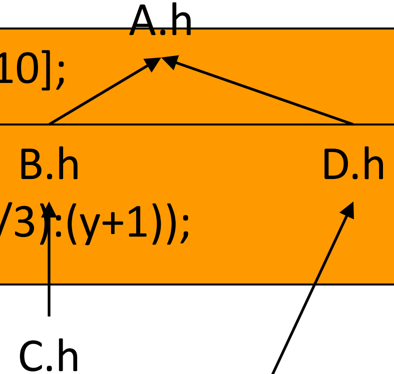
Uvodi novi simbol N. Svako njegovo pojavljivanje će biti zamenjeno sa 10

Kompajler će videti: int a[10];

Kompajler će videti:
int c = (((x/3)>=(y+1))?(x/3):(y+1));

U... pojavljivanje max(?,?) će biti zamenjeno sa odgovarajućim tekstom uz automatski prenos parametara

Uvodi simbol _A_h u tabelu simbola preprocesora



X.cpp

- Razmotrimo scenario

Neka je realizovana klasa A (A.h i A.cpp) i neka se njena definicija (A.h) direktno ili indirektno uključuje u više fajlova. Kompajler će to protumačiti kao da postoji više definicija iste klase (što je strogo zabranjeno) i prijaviće grešku.

- Problem se rešava upotrebom **#ifndef**/**#endif** direktiva:

```
#ifndef _A_h
#define _A_h
 ... // Kod header fajla
     // će biti prosleđen kompajleru
     // samo pri prvom pojavljivanju #include
#endif
```

- Sličan koncept se može upotrebiti pri projektovanju platform-dependant koda:

```
#ifdef Windows
 ... // Windows-dependent code
#elseif defined(Linux)
 ... // Linux-dependent code
#endif
```

V-2: Kompajler

A.cpp

```
int a = 3;

void f() {
   ...
}
```

B.cpp

```
extern int a;
void f();

void g() {
   ...f()...
   ...a...
}
```

a i f se moraju deklarisati pre upotrebe

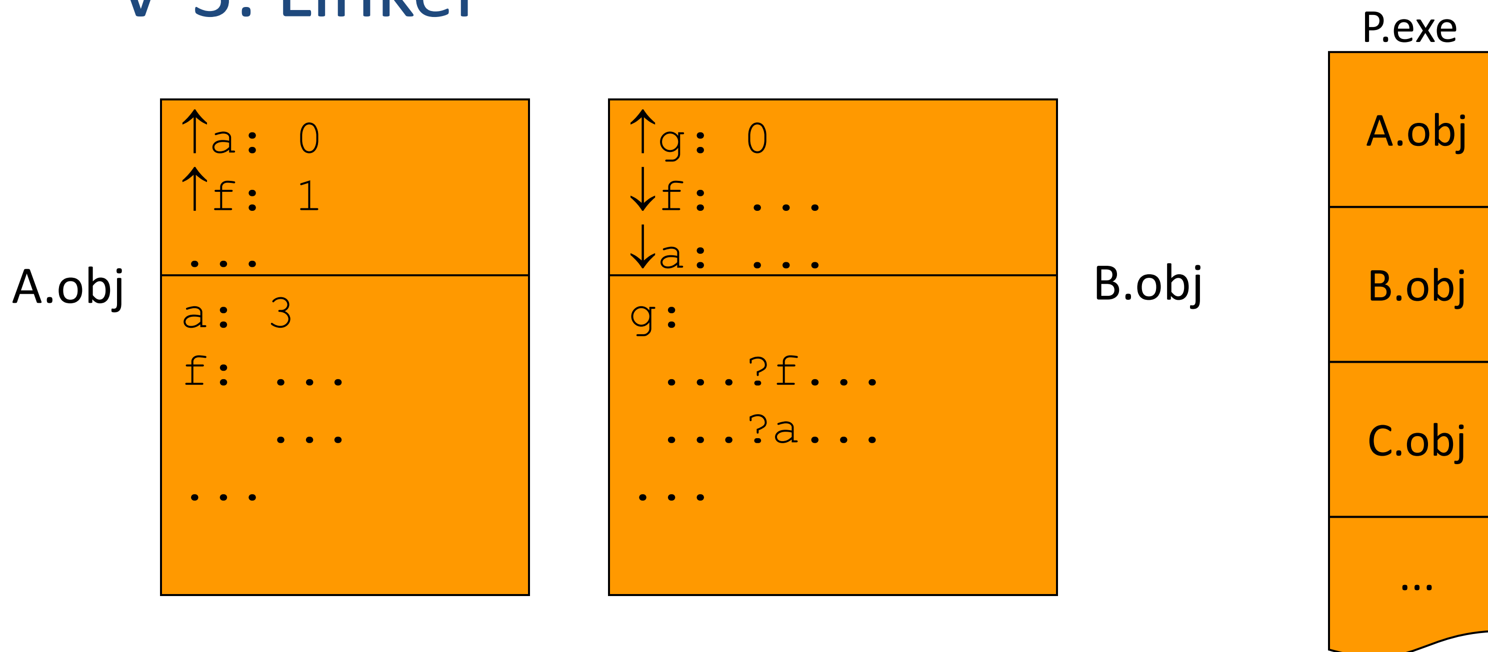
A.obj

```
↑a: 0
↑f: 1
...
a: 3
f: ...
   ...
...
```

B.obj

```
↑g: 0
↓f: ...
↓a: ...
g: ?
....?f...
 ...?a...
...
```

V-3: Linker



- Najčešće greške pri linkovanju:
  - Undefined symbol (dlopen error). Greška je obično konfuznog oblika i često prijavljuje simbole koji se i ne koriste u kodu. Najčešći uzrok: linkeru nisu prosleđene odgovarajuće biblioteke.
  - Multiple symbol definition. Najčešći uzrok: postoji više definicija istog objekta u više .cpp fajlova

VI: Klase i objekti

- Klasa je strukturni tip koji obuhvata:
  - podatke koji opisuju stanje objekta klase **(polja klase)**
  - funkcije namenjene definisanju operacija nad podacima klase **(metode klase)**
- Klasa je realizacija apstrakcije a jedan njen primerak se naziva **objektom** te klase
- Članovi klase mogu biti:
  - **`private:`** (podrazumevano)
    - dostupni samo unutar klase
    - mogu im pristupiti prijateljske f-je ili klase
  - **`public:`**
    - dostupni u klasi i spolja bez ograničenja
  - **`protected:`**
    - dostupni u datoj i izvedenoj klasi
- Redosled navođenja prava pristupa je proizvoljan

- Polja klase ne mogu biti tipa klase u kojoj se nalaze ali mogu biti referenca ili upućivač
- Sa klasama je podrazumevano moguće:
  - definisanje objekata, pokazivača i referenci na objekte i nizova objekata klase
  - dodeljivanje vrednosti jednog objekta drugom (**=**)
  - neposredno pristupanje poljima i metodama (**.**)
  - posredno pristupanje poljjima i metodama preko pok (**->**)
  - korišćenje objekata kao argumenata i/ili povratne vrednosti f-ja (može i preko pokazivača/upućivača)
- Unutar svake metode klase postoji sakriveni pokazivač **this**
  - ako je definisana klasa **X**, **this** je onda uvek tipa **X* const**
  - **this** nam omogućava da eklspicitno pristupimo članicama klase, mada je to suštinski nepotrebno je se radi implicitno
  - najčešće se koristi kao povratna vrednost neke metode klase

- Primer definisanja klase, metode, polja i **this** pokazivača

```
class Kompl {
  public:
     Kompl zbir(Kompl);
     Kompl razlika(Kompleksni);
     float re(); float im(); // poljima klase ne
           pristupamo direktno vec preko metoda
  private: // Polja klase su zasticena
     float real, imag;
};
Kompl Kompl :: zbir(Kompl c) {
  Kompl t;
  // Umesto this->real moze samo real
  t.real = this->real + c.real;
  t.imag = this->imag + c.imag;
  return t; }
```

VI-1: Konstruktori (podr., kopije, konverzije)

- Konstruktor je specijalna metoda klase koja definiše početno stanje objekta koji se kreira (inicijalizuje polja)
  - nosi isto ime kao i klasa
  - nema tip rezultata (čak ni void)
  - ima proizvoljan broj argumenata
  - broj konstruktora je proizvoljan
- Opšti oblik konstruktora je:

```
Klasa (parametri) : inicijalizator, ..., telo
 gde je inicijalzator: polje (izraz, ... , izraz)
```

- Primer:

```
Kompl :: Kompl () : real(0), imag(1) {} //ili
Kompl :: Kompl () { real = 0; imag = 1; }
Kompl :: Kompl (int i) : real(i), imag(i) {} //ili
Kompl :: Kompl (int i) { real = i; imag = i;}
```

- Kompajler svaki konstrukor koji nema ni jedan argument (ili čiji svi argumenti imaju podrazumevane vrednosti) proglašava za **podrazumevani**. Ovaj konstruktor se poziva implicitno pri svakom kreiranju instance klase!
- Ukoliko ne postoji ni jedan konstruktor automatski se generiše podrazumevani sa praznim telom. Ovaj konstruktor je u stanju da ispravno inicijalizuje samo ona polja klase koja imaju svoje podrazumevane konstruktore.
- Pored podrazumevanih postoje i konstruktori:
  - **kopije**
    - za argument ima upućivač na isti tip kao i klasa kojoj konstruktor pripada
    - poziva se implicitno pri inicijalizaciji objekta, kada se objekat prosleđuje kao argument u f-ju ili kada f-ja vraća objekat
  - **konverzije**
    - odredišni tip konstruktora mora biti klasa dok argumenti mogu biti bilo kog tipa
    - poziva se implicitno, izuzev ako ga ne definišemo kao **explicit**

- Primer:

```
class Counter {
public:
   Counter(int);
   ...
};
```

Klasa ima jedan konstruktor. On NIJE podrazumevani jer ima argument koji nema podrazumevanu vrednost. Kompajler neće generisati podr. kosntrukor

```
class Clock {
private:
   Counter c;
};
```

Ova klasa nema eklsplicitni konstruktor pa će kompajler implicitno generisati podrazumevani sa praznim telom:
`public Clock::Clock () : c() {}`
Kako klasa Counter ne poseduje podrazumevani konstruktor , kompajler prijavljuje grešku

```
class Clock {
public:
   Clock() {}
private:
   Counter c;
};
```

U ovom slučaju iako je sprečeno generisanje podrazumevanog konstruktora klase Clock, opet imamo istu grešku jer će kompajler svejedno pokušati da implicitno alocira prostor za c

VI-2: Destruktor

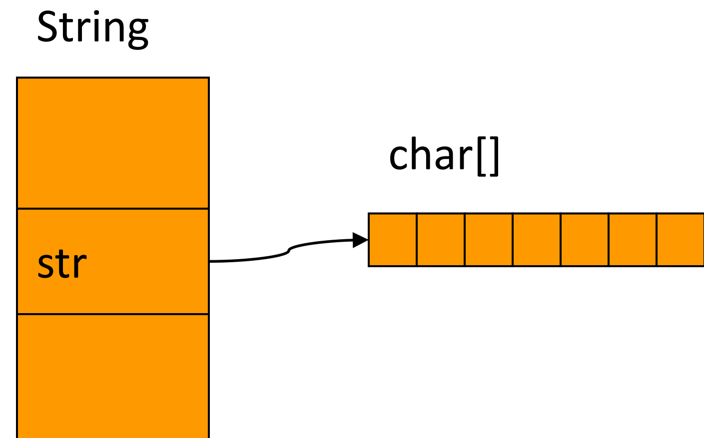
- Destruktor je specijalna metoda klase koja implicitno uništava objekat na kraju njegovog životnog veka
  - nosi isto ime kao i klasa, uz znak ~ isped imena
  - nema tip rezultata (čak ni void)
  - ne može imati argumente
  - može postojati samo jedan
- Ukoliko nije definisan, destruktor sa praznim telom se implicitno generiše i uništava sva polja klase koja imaju odgovarajuće podrazumevane destruktore

```
class X {
public:
... };
void main() {
X x;
} // Ovde se poziva destruktor objekta x
```

- Pravilo je da kada baratamo sa dinamičkim podacima moramo da definišemo svoj destruktor

```
class String {
public:
   String () : str(0) {}
   String (char*);
  ~String () { delete str; }
   ...
private:
   char* str;
};

String::String(char* s) {
   if (str = new char[strlen(s)+1]) strcopy(str,s);
}
```



VI-3: Statička polja i metode klase

- **Statička (zajednička) polja** klase su dostupna svim objektima klase. Statičko polje ima samo jednu instancu bez obzira na broj kreiranih objekata klase.
  - deklaracija statičkog polja se nalazi u telu klase dok se njegova definicja (inicijalizacija) uvek vrši van nje (pre poziva **`main()`**). Razlog je što statičko polje treba da postoji i pre nego što se deklarišu objekti tipa klase.
  - upotreba statičkih polja omogućava da objekti iste klase dele neki podatak i/ili da preko njega međusobno “komuniciraju”. Ona se ponašaju isto kao i globalne statičke promenljive sem što su enkapsulirane u klasu (mogu biti **public, private, protected**) i imaju doseg klase (pristup putem **::**).

VI-3: Statička polja i metode klase

- **Statičke (zajedničke) metode** klase ne pripadaju konkretnom objektu klase već klasi kao tipu podatka
  - shodno prethodnom one nemaju pokazivač **this** pa ne mogu pristupati nestatičkim poljima klase
  - podležu enkapsulaciji i imaju doseg klase kojoj pripadaju (pristup putem **::**)

```
class Abc {
    static int a;    // Zajednicko polje
    int b;           // Pojedinacno polje
public:
    static int f(); // Zajednicke metode
    static void g (Abc, Abc*, Abc&);
    int h (int);     // Pojedinacne metode
};
// Definicija i test na sledecem slajdu
```

```
int Abc::f () {
     int i = a;    // OK, staticko polje
     int j = b;    // Err! implicitno se poziva this->b
     return i+j;   // dok this ne postoji!
}
void Abc::g(Abc x, Abc* y, Abc& z) {
     int i = x.b;  // Pristup pojedinacnim poljima nekog
     int j = y->b; // objekta samo ako je on argument
     z.b = i +j;   // f-je
}
int Abc::h (int x) {
     return (a+b)*x; // OK, this postoji
}
int Abc::a = 55;         // Inicijalizacija pre main ()
void main () {
     int p = Abc::f ();  // OK, staticka
     int q = Abc::h (5); // Err! Mora konkretan objekat
     Abc k;              // Kreira se objekat
     int r = k.h ();}    // OK, levi operand je objekat
```

VI-4: Prijateljske funkcije klasa

- Prijateljske f-je nisu članovi klase, ali imaju privilegovan pristup njenim poljima
  - deklarišu se unutar klase (svejedno dali u **public** ili **private** delu)
  - ne poseduju pokazivač **this**, pa mogu da operišu samo na konkretnim objektima koji im se prosleđuju kao argumenti
  - mogu pristupiti i privatnim poljima klase
  - mogu biti istovr. prijatelji više klasa i globalnog su karaktera
- Deklaracija su oblika:

```
friend tip funkcija (argumenti) {}; // Standardno
friend ime_klase; // Sve prijateljske f-je od ime_klase
                  // automatski postaju prijatelji klase
                  // gde je naredba navedena
friend class ime_klase; // Isto kao prethodno sem sto
                        // dozvoljava da ime_klase nije
                        // ni definisano ni deklarisano
```


Marko Marković

- Poređenje prijateljskih f-ja i metoda klase

```
class Alfa {
        int x;
public:
        void p (int n) { x = n; }
        int q () { return x; }
        friend void r (int n, Alfa& a) { a.x = n; }
        friend int s (Alfa a) { return a.x}
};
void main () {
        Alfa a;
        a.p (55);           // Smestanje metodom
        int i = a.q (); // Uzimanje metodom
        r(55, a);          // Smestanje prijateljskom f-jom
        int j = s (a); // Uzimanje prijateljskom f-jom
}
```

VII: Korisni linkovi

- C++ dokumentacija sa primerima:
  http://www.cplusplus.com/

  http://www.cprogramming.com/

  http://en.cppreference.com/w/cpp
- FAQ / Forumi

  http://www.parashift.com/c++-faq-lite/
  http://groups.google.com/group/comp.lang.c++.moderated/topics?pli=1

  http://stackoverflow.com/
- Predavanja i vezbe sa ETF-a

  http://oop.etf.rs/index.html
  http://rti.etf.bg.ac.rs/rti/ir2oo1/index.html