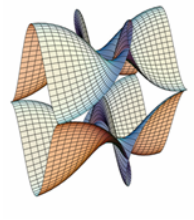

Prirodoslovno-matematički fakultet
Matematički odsjek
Sveučilište u Zagrebu

# RAČUNARSKI PRAKTIKUM I

## Vježbe 07 - Podstrukture, const, reference

v2018/2019.


Sastavio: Zvonimir Bujanović

# Podstrukture

Član neke strukture može biti i druga struktura.

```
struct stogStringova
{
    ...
};

struct Automobil
{
    string registracija;
    stogStringova vlasnici;
    int godinaProizvodnje;
};
```

# Podstrukture: inicijalizacija

Podstruktura koja je član neke strukture se inicijalizira prije ulaska u konstruktor te strukture.

Automatski se poziva defaultni konstruktor za podstrukturu.
Ako želimo pozvati neki drugi konstruktor, to radimo ovako:

```
struct stogStringova
{
    stogStringova( int velicina ) { ... }
};

struct Automobil
{
    int brojVlasnika;
    stogStringova vlasnici;

    Automobil() : vlasnici( 1 ), brojVlasnika( 1 ) { ... }
    Automobil( int brVl )
        : vlasnici( brVl ), brojVlasnika( brVl ) { ... }
};
```

# Podstrukture: inicijalizacija

C++11 – Podstruktura se može inicijalizirati i ovako:

```
struct stogStringova
{
    stogStringova( int velicina ) { ... }
};

struct Automobil
{
    int brojVlasnika = 1;
    stogStringova vlasnici = stogStringova( 1 );

    Automobil() { ... }   // dogodi se gornja inicijaliz.
    Automobil( int brVl ) // ne dogodi se gornja inicijaliz.
        : vlasnici( brVl ), brojVlasnika( brVl ) { ... }
};
```

Ova inicijalizacija se dogodi ako ne navedemo popis konstruktora za podstrukture prije ulaska u konstruktor nadstrukture.

# Ugniježđene strukture

Unutar strukture možemo deklarirati drugu, ugniježđenu strukturu.

```
struct Automobil
{
    struct Motor
    {
        int snaga;
        double obujam;
        Motor( int s, double o ) { snaga = s; obujam = o; }
    };

    Motor m;
    int brojVrata;
    Automobil() : m( 90, 1.4 ) { ... }
    Automobil( int s, double o ) : m( s, o ) { ... }
};
```

Ovo je korisno kada ugniježđena struktura ima smisla samo unutar nadstrukture.

# Ugniježđene strukture

Izvan strukture `Automobil` moguće je deklarirati varijable tipa `Automobil::Motor`.

```
Automobil yugo( 45, 1.2 );
Motor tdi( 120, 1.9 ); // !!! krivo, compile error !!!
Automobil::Motor turboDiesel( 120, 1.9 ); //  OK

yugo.m = turboDiesel;

cout << yugo.m.snaga;
cout << yugo.brojVrata;
```

# Zadatak 1

- Napravite implementaciju atp-a `list` pomoću polja; lista treba moći čuvati max. 100 elemenata proizvoljnog tipa.
- Unutar liste treba postojati ugniježđena struktura `iterator` tako da se može raditi sljedeće:

```
list<int> L;
L.push_back( 5 ); L.push_back( 7 );

list<int>::iterator li = L.begin();
while( !li.isEqual( L.end() ) )
{
    cout << li.data() << " ";
    li = li.next();
}
```

Uputa: neka `iterator` sadrži pokazivač na tip iz template-a.

# const

Varijable označene kao `const` prilikom deklaracije kasnije nije moguće mijenjati.

Takve varijable moramo inicijalizirati.

```
const int a;      // !!! compile error !!!
const int b = 5; //  OK
```

U `const` varijable se ne može "pisati", ali ih se može "čitati".

```
const int a = 5;
int b = 3;

a = b; // !!! compile error: pokusaj pisanja u const !!!
b = a; //  OK
```

# Pokazivači na const

Pokazivač na konstantne podatke tipa `Type` deklariramo ovako:

```
const Type *ptr;
```

Ovdje su konstantni podaci na koje pokazuje `ptr`, a ne i sam `ptr`!
Zato nije potrebno inicijalizirati `ptr`.

```
char nesto[] = "rp1";

char *s1 = nesto;         // inicijaliziramo obicni pokazivac
const char *s2 = nesto; // inicijaliziramo pokazivac na const

s1[1] = 'A'; // OK
s2[1] = 'A'; // greska, mijenjam podatke na koje pokazuje s2

s1 = s2; // greska, preko s1 se mogu mijenjati podaci u s2!
s2 = s1; // OK (s2 nije konstantan, nego samo podaci)
```

# Pokazivači na const

Ponekad funkcije (poput `c_str()`) vraćaju pokazivače na const.

```
void ispis1( char *s ) { ... }
void ispis2( const char *s ) { ... }

string s = "rp1";

char *s1;
const char *s2;

s1 = s.c_str(); // greska, c_str() vraca const char *
s2 = s.c_str(); // OK

ispis1( s.c_str() ); // greska
ispis2( s.c_str() ); // OK
```

# const funkcije članice

Funkcije članice strukture koje ne modificiraju objekt moguće je označiti sa `const` prilikom deklaracije.

Takve funkcije možemo pozivati i na const objektima.

```
struct MyString
{
    int size;
    char *data;

    MyString() { size = 0; data = nullptr; }

    int length() const { return size; }
    const char *c_str() const { return data; }
    void append( const char *s ) { ... }
};

const MyString s;    // OK, inicijalizaciju radi konstruktor
int i = s.length(); // OK, length je const funkcija
s.append( "abc" );  // !!! compile error !!! append nije const
```

# const funkcije članice

Dozvoljeno je imati dvije varijante iste funkcije: jednu koja radi s const objektima, a drugu koja radi s ne-const objektima.

```
struct MyStruct
{
    string f() const { return "ja sam const"; }
    string f() { return "ja nisam const"; }
};

const MyStruct s1;
MyStruct s2;

cout << s1.f(); // "ja sam const"
cout << s2.f(); // "ja nisam const"
```

Ako funkcija ima istu implementaciju i za const i za ne-const objekte:

- dovoljno je napisati samo const-varijantu;
- tada će ona biti pozivana u oba slučaja.

# "Varijabilni" parametri funkcija

U C-u, parametri funkcijama se šalju po vrijednosti, tj. funkcija ne može promijeniti varijablu koju joj šaljemo kao parametar.

To zaobilazimo slanjem pokazivača na varijablu.

```
void f( int *p )
{
    *p = 3;
}

int main()
{
    int x = 5;
    f( &x );
    cout << x; // ispisuje 3

    return 0;
}
```

# Reference

Uz "obične" varijable i pokazivače, C++ uvodi reference.

- Referenca na varijablu `x` ≡ drugo ime za varijablu `x`.
- Referenca se uvijek mora inicijalizirati. Nakon inicijalizacije, referenca ostaje zauvijek "zavezana" za istu varijablu.
- Koristiti referencu je potpuno isto kao koristiti originalnu varijablu, u svim mogućim situacijama.
- Dakle, referenca dijeli istu adresu s varijablom.

```
int a = 5, b = 3;
int &r = a; // OK, r je referenca (drugo ime) za a
int &t = 5; // !!! compile error, nema varijable !!!
int &s;     // !!! compile error, nema varijable !!!

cout << r; // ispisuje 5 (tj. vrijednost od a)
r = b;     // u r (tj. u a) spremi vrijednost od b
cout << r << " " << a; // ispisuje 3 3
a = 8;
cout << r << " " << a; // ispisuje 8 8
```

# Reference kao parametri funkcijama

Ako je parametar funkcije referenca, onda promjena vrijednosti te reference unutar funkcije zapravo mijenja varijablu s kojom je funkcija pozvana.

```
void f( int &a ) // a je referenca na varijablu iz main-a
{
    a = 7;
}

int main()
{
    int x = 5;
    f( x );     // kao da pise: int &a = x;
    cout << x; // ispisuje 7

    f( 5 ); // !!! compile error, nema varijable !!!

    return 0;
}
```

# Reference na const

Pomoću reference na const ne možemo mijenjati vrijednost referirane varijable.

```
int a = 5;
const int &r = a;

cout << r; // ispisuje 5
r = 7; // !!! compile error, r je referenca na const !!!

a = 7; // OK, a nije const varijabla
cout << r; // ispisuje 7
```

Funkcije često imaju parametre koji su reference na const.

- Ako funkcija prima referencu, onda se ne događa kopiranje cijelog objekta. Zato je poziv funkcije brz i za velike objekte.
- Ako ne želimo da funkcija može mijenjati referiranu varijablu, označit ćemo referencu sa const.

# Referenca kao povratna vrijednost funkcije

Funkcija može vraćati referencu.

```
int &prviElementVektora( vector<int> &X ) {
    return X[0];
}

int main()
{
    vector<int> V( 10 );

    cout << V[0]; // ispise 0
    prviElementVektora( V ) = 5; // !!! OK !!!
    cout << V[0]; // ispise 5

    return 0;
}
```

Što bi se dogodilo da funkcija nije primila referencu, nego “običnu varijablu”?

# Referenca kao povratna vrijednost funkcije

Oprez: ne smijemo vraćati referencu na lokalnu varijablu u funkciji!

```
int &proba( int *a ) {
    int temp = 5; *a = 5;

    // Iduca linija bi srusila program prilikom izvrsavanja.
    // return temp; // krivo (warning), temp je lokalna!

    return *a; // OK, *a zivi i izvan funkcije.
}

int main()  {
    int x = 3;
    int y = proba( &x );  // OK, y = 5;
    int &t = proba( &x ); // OK
    ++t;
    cout << x; // ispise 6

    return 0;
}
```

# Reference: bonus pitanja

1. Zbog čega copy-konstruktor prima referencu, a ne "običnu" varijablu i zašto je to referenca na const?

2. Neka su `li` i `si` iteratori u listi i set-u, redom. Prilikom izvršavanja `*li` i `*si` se zapravo poziva neka funkcija (označena dolje sa `retrieve`). Objasnite zašto ta funkcija treba vraćati referencu u slučaju liste, a referencu na const u slučaju set-a.

```
cout << *li;   ⇔   cout << retrieve(li);
```