



Primenjeno softversko inženjerstvo

Napredni C kurs

(Strukture podataka)

Struktura C programa

- Konstante i tipovi podataka
- Promenljive (lokalne, globalne, eksterne...)
- Izrazi (if, while, for...)
- Operatori (+, -, *, /, ~...)
- Funkcije sa argumentima
 - Prenos po vrednosti
 - Prenos po adresi
 - Prenos po referenci (C#)
- Makro izrazi
- Preprocesorske direktive
- Biblioteke
- Komentari

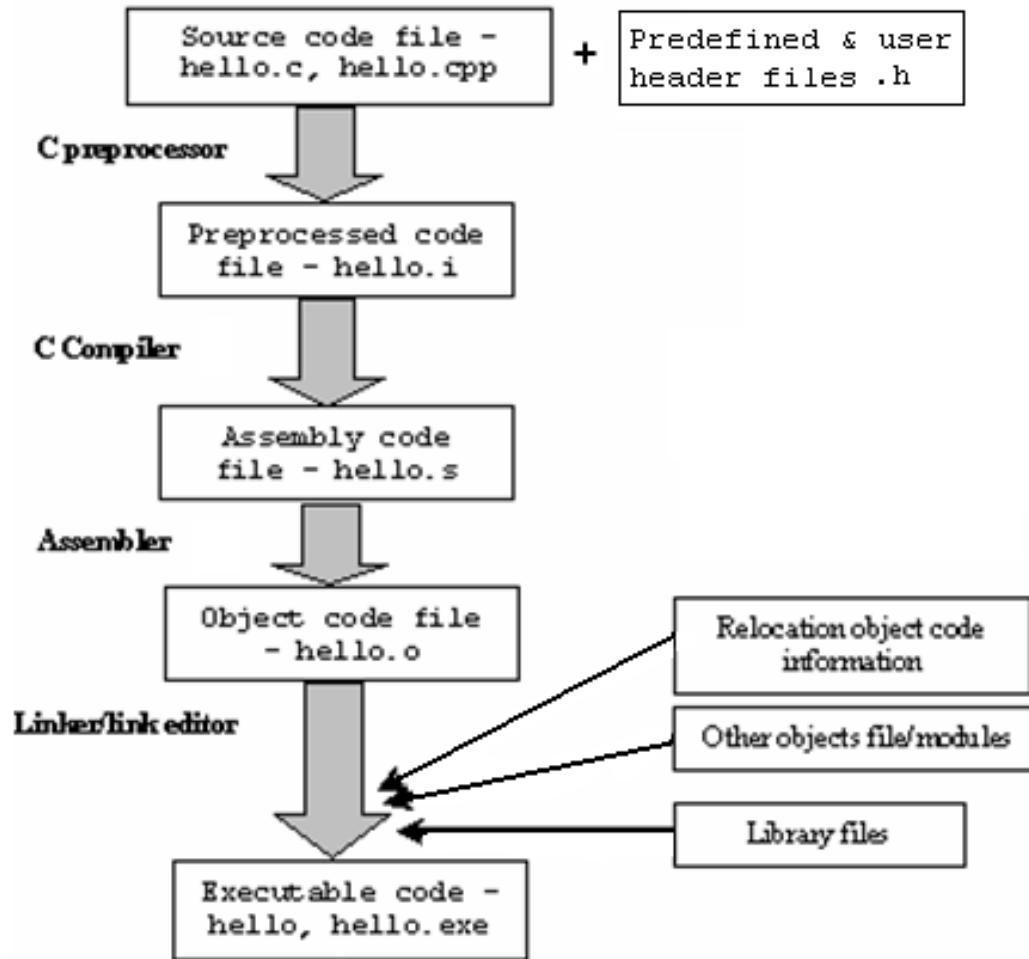
```
#include <stdio.h>
#include "mydef.h"

#define ....      /* lokalne definicije */
int gvInt;          // globalna promenljiva
extern ...          /* referencia na gvar */
int func1(...);    /* redeklaracija */

void main( void )
{
    int i;          /* lokalna promenljiva */
    izraz 1;        /* komentar */
    func1(i, &i);  /* poziv funkcije */
    izraz 2;
    ....
}
int func( int k, int *pk )
{
    izraz3;        /* telo funkcije */
    return 0;
}
```

Prevođenje C programa

- file_name.c
 - C izvorni kod
- file_name.h
 - C header file
- file_name.i
 - Preprocesirani kod
- file_name.s, .asm
 - Asemblerski kod
 - Asemblerski kod koji se mora preprocesirati
- file_name.o, .obj
 - Objektni kod
- file_name.exe
 - Izvršni kod



Faze prevodenja C programa

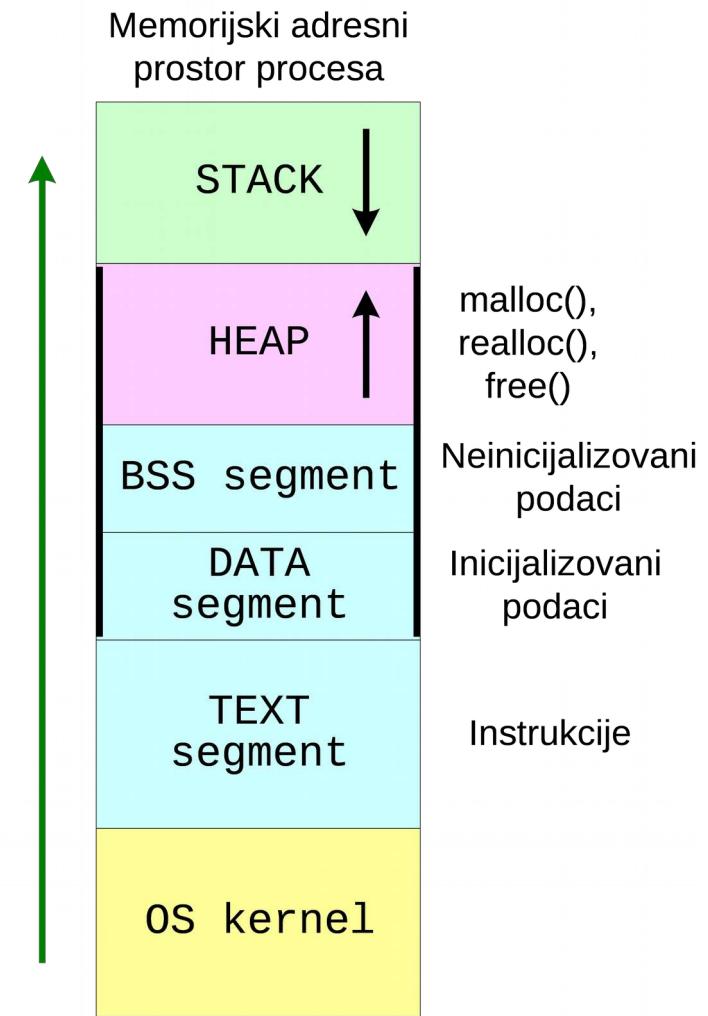
- Izvorni kod -> Asemblerski kod -> Objektni kod -> Izvršni kod
- Preprocesor – prvi prolaz prevodenja C koda
- Kompajler – drugi prolaz, generiše asemblerski kod
- Asembler – generiše objektni kod i asemblerski listing
- Linker – final faza gde se
 - Više .obj i .lib modula kombinuju u jedan izvršni (.exe)
 - Rešavaju reference na spoljne simbole
 - Vrši konačno dodeljivanje adresa funkcijama/promenljivim (relokacija)
- U IDE kompjajlerima ove procedure su integrisane

Programski model

- U krajnjoj posledici programski model je
 - Organizacija promenljivih i funkcija, i načina međusobnog povezivanja, u okviru izvršnog programa
 - Iako zavisan od HW platforme, sličan u većini implementacija
- Postaje značajan u krajnjim fazama C prevodenja, zato pod programskim modelom podrazumevamo
 - Konvencije generisanja asemblerorskog koda
 - Memorijska alokacija promenljivih
 - Registrarske konvencije i korišćenje stack-a
 - Garancija međusobne kompatibilnosti
 - Povezivanje programa različitih kompjlera
 - Aplikativnih programa i operativnih sistema

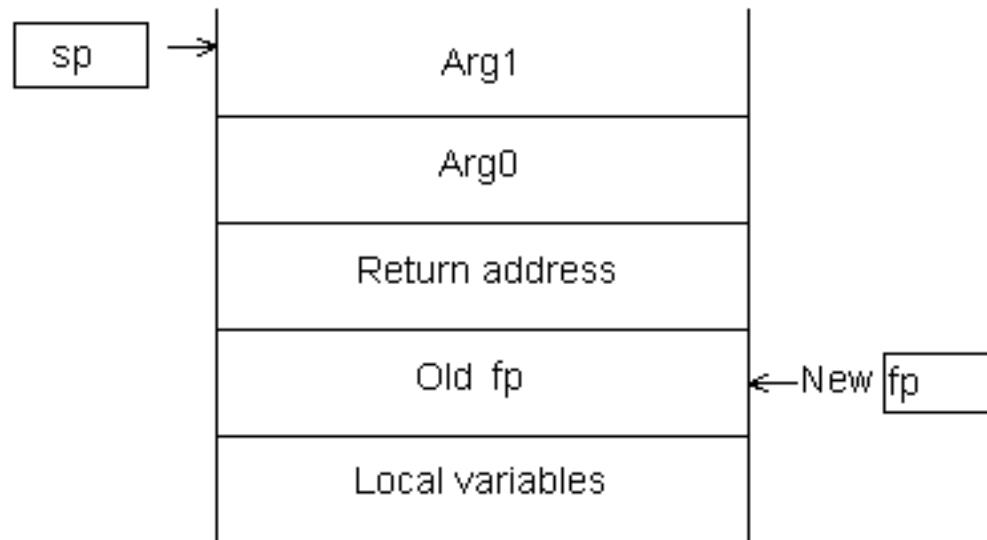
Memorijski segmenti programa

- Svaki program dobija svoj prostor pri punjenju
- TEXT sekcija može biti deljena
 - reentrant code – programs
- DATA – non-zero init global i static
 - .rdata – read only (const)
- BSS (Block Started by Symbol) – u izvršavanju DATA, u .exe ga nema
- HEAP – dinamička memorija
 - Pristup samo poreko pokazivača
 - Kontrola opsega (kompajler, runtime)
- STACK – lokalne promenljive, poziv funkcija i prenos argumenata
 - Stack frame, stacksize



Korišćenje stack-a pri pozivu funkcija

- Stack frame – osnovna struktura
 - Formira se i briše pre/po svakom pozivu funkcije na tekućoj lokaciji stack-a
 - Sadrži argumente, povratnu adresu i lokalne promenljive



Izgled stack-a pri pozivanju funkcija

```
int main()
{
    a();
    return 0;
}
```

```
int a()
{
    b();
    c();
    return 0;
}
```

```
int b()
{ return 0; }

int c()
{ return 0; }
```

Higher
memory

Frame
for
main()

Lower
memory

Frame
for
main()

Frame
for
a()

main()
calls a()

Frame
for
main()

Frame
for
a()

Frame
for
b()

a() calls
b()

Frame
for
main()

Frame
for
a()

return
from b()

Frame
for
main()

Frame
for
a()

Frame
for
c()

a() calls c()

Frame
for
main()

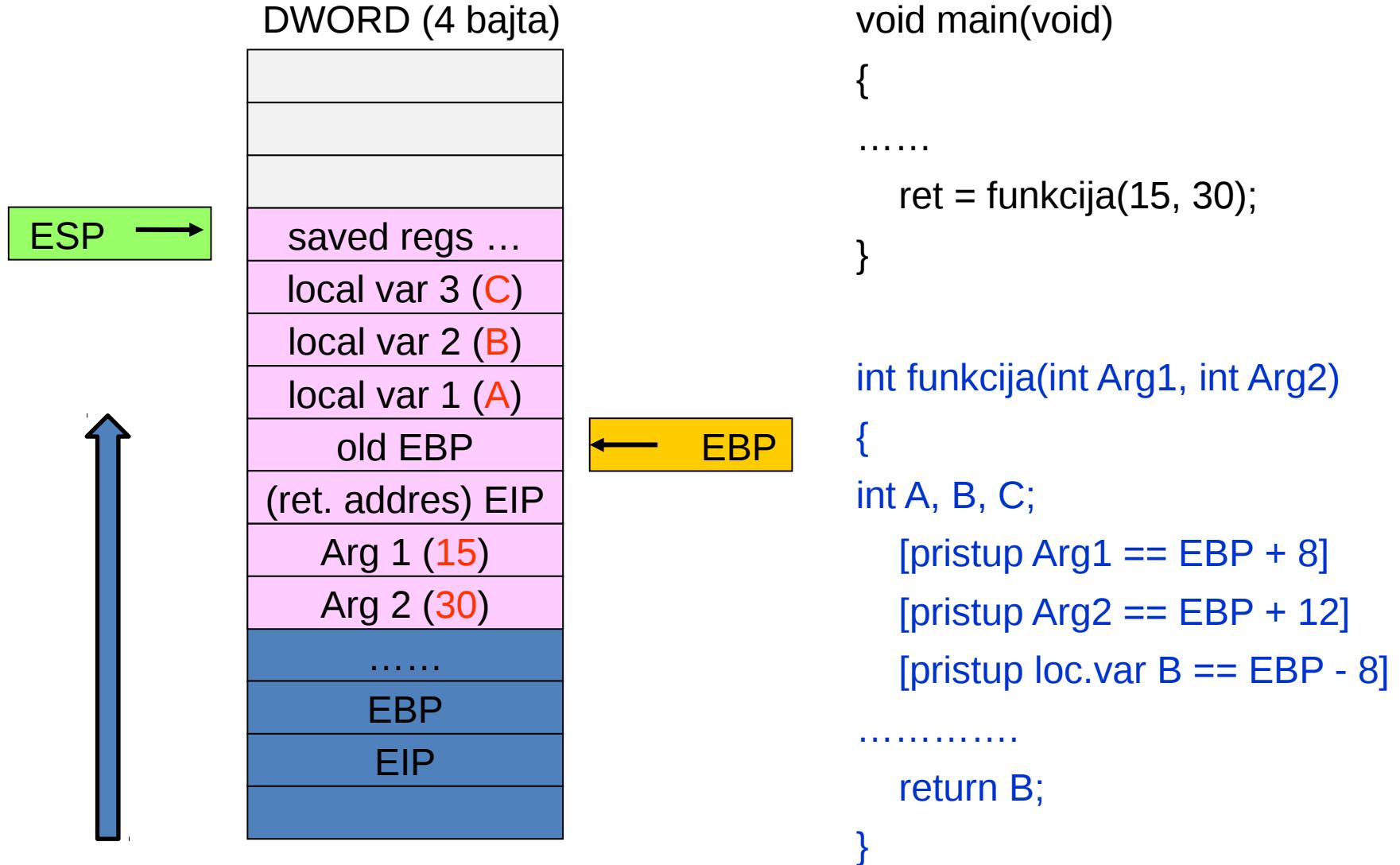
Frame
for
a()

return from
c()

Frame
for
main()

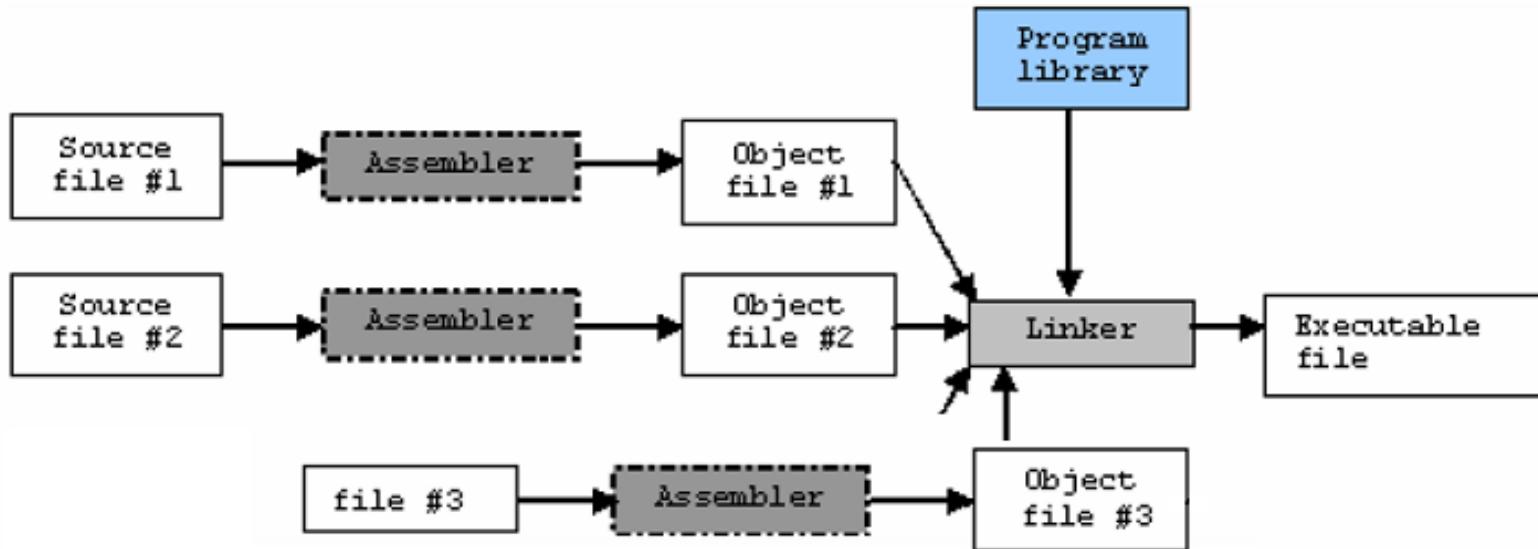
return from
a()

Stack frame



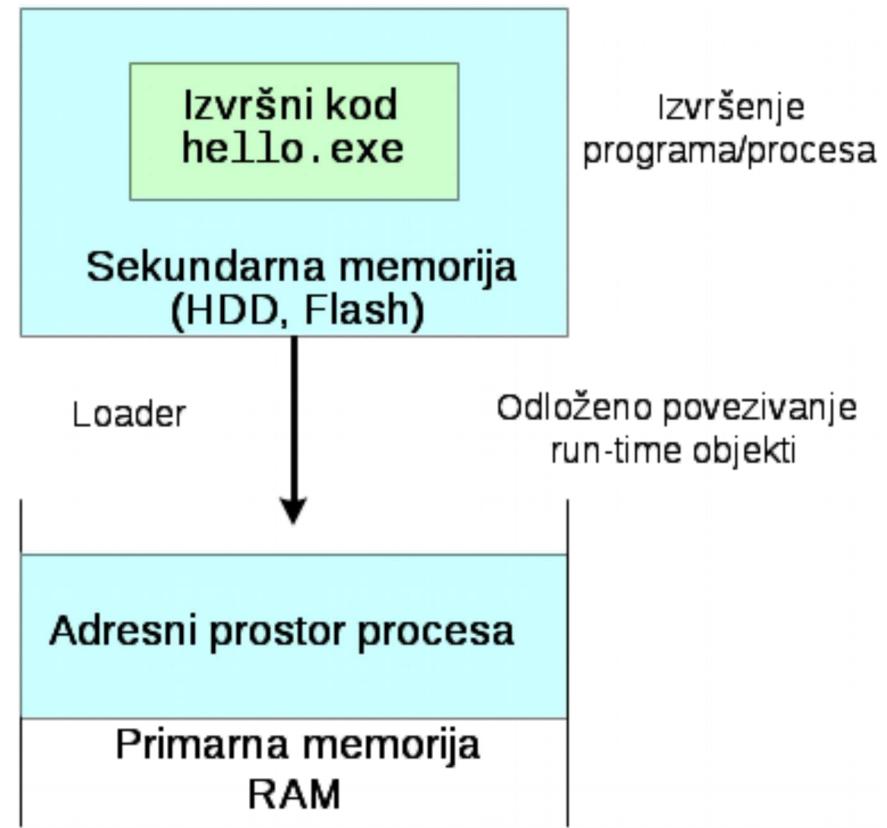
Linker

- Povezuje u izvršni kod, rešavanjem međusobnih adresnih referenci
 - Statičko – sve potrebno u izvršni fajl
 - Dinamičko – deljeni izvršni moduli (.dll)



Punjač (Loader)

- Pre izvršavanja se puni u operativnu memoriju
 - Funkcija OS
 - Sa diska
 - Sve se kopira
 - Iz Flash-a
 - Sve se kopira (max.var)
 - Samo inicijalizovane promenljive (min.var)
- Dinamičko povezivanje
 - Deffered linking
 - Run-time moduli/biblioteke



Proces punjenja

- Verifikacija izvršnog programa i računanje memorijskih zahteva
- Verifikacija raspoložive memorije i prava pristupa
- Alokacija potrebne memorije i prenos sa sekundarne memorije
- Formiranje sekcija podataka (DATA + BSS + Heap sekcija)
- Inicijalizacija stack-a i argumenata za main() funkciju
- Pozivanje main() funkcije

Direktive C Preprocesora

- Uključivanje drugih datoteka u fajl za prevodenje
 - `#include <stdlib.h>`, `#include "MyDef.h"`
 - Koristiti / (ne \) npr. `#include "Common/MyDef.h"`
- Definicija konstanti, tipova podataka i makroa
 - `#define PI 3.14`
 - `#define AREA(a,b) ((a)*(b))`
- Kontrola prevodenja (conditional compilation)

```
#if !defined(NULL)
    #define NULL 0
#endif
```

```
#ifdef DEBUG
    printf("Var x= %d", x);
#endif
```

Predefinisani makroi

Simbolička konstanta	Opis
<code>__DATE__</code>	Datum prevodenja
<code>__LINE__</code>	Broj linije u .c datoteci
<code>__FILE__</code>	Ime datoteke izvornog koda
<code>__TIME__</code>	Vreme prevodenja
<code>__STDC__</code>	Označava ANSI C kompatibilnost

Tipovi podataka u C

Type Name	Bytes	Other Names	Range of Values
int	4	signed	-2,147,483,648 to 2,147,483,647
unsigned int	4	unsigned	0 to 4,294,967,295
_int8	1	char	-128 to 127
unsigned _int8	1	unsigned char	0 to 255
_int16	2	short, short int, signed short int	-32,768 to 32,767
unsigned _int16	2	unsigned short, unsigned short int	0 to 65,535
_int32	4	signed, signed int, int	-2,147,483,648 to 2,147,483,647
unsigned _int32	4	unsigned, unsigned int	0 to 4,294,967,295
_int64	8	long long, signed long long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned _int64	8	unsigned long long	0 to 18,446,744,073,709,551,615
bool	1	none	false or true
char	1	none	-128 to 127 by default, 0 to 255 when compiled by using /_J
signed char	1	none	-128 to 127
unsigned char	1	none	0 to 255
short	2	short int, signed short int	-32,768 to 32,767
unsigned short	2	unsigned short int	0 to 65,535
long	4	long int, signed long int	-2,147,483,648 to 2,147,483,647
unsigned long	4	unsigned long int	0 to 4,294,967,295
long long	8	none (but equivalent to _int64)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8	none (but equivalent to unsigned _int64)	0 to 18,446,744,073,709,551,615

Promenljive

- Deo memorije koji čuva određeni tip podataka
 - tip_pod ImePromenljive;
 - sizeof(ImePromenljive) == sizeof(tip_pod)
- Tipovi promenljivih:
 - lokalne – vidljive unutar jedne funkcije
 - Smeštene na stack-u
 - globalne – definisane iznad tela funkcije
 - extern – spoljne (drugi modul)
 - static – dostupne samo funkcijama u jednom modulu
 - const – konstante (read only)
 - register – čuvaju se u registru
 - volatile – podložna promeni iz prekida

Izvedeni tipovi podataka

- Direktive: struct, union, typedef, enum
- Sve izvedene tipove treba označiti typedef-om
- Prvi član enum-a uvek mora biti inicijalizovan
- Strukture su zgodan način za grupisanje i prenos podataka između funkcija
 - $\text{sizeof(struct)} == \sum_i \text{sizeof(član}_i)$, za pack(1)

```
Pris typedef struct {
    uint16_t      x;
    uint16_t      y;
} gdiPoint_t;

typedef enum { ILL = -1, GOOD, ... } boxStatus_t;

typedef struct listNode_t {
    struct listNode_t *next;
    int              data;
} listNode_t;
```

Izvedeni tipovi podataka

- union direktiva definiše tip podataka gde članovi dele isti prostor za smeštanje (storage space)
 - `sizeof(union) = sizeof(najduži član)`

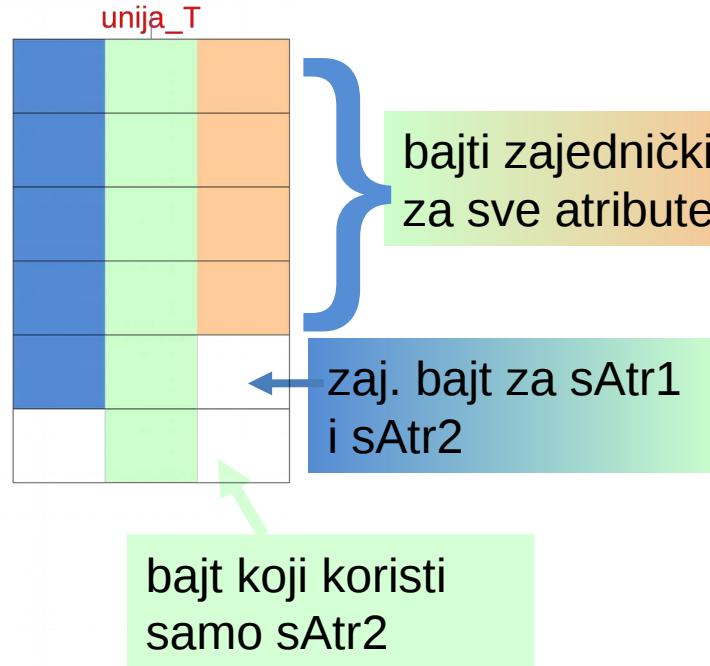
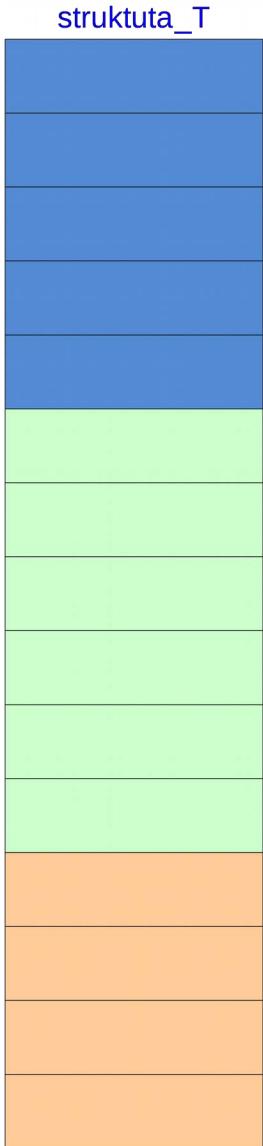
```
union sample
{
    int p;
    float q;
};

union sample content = {234};
    ili
union sample content = {24.67};

    pristip članovima unije

content.p = 37;
content.q = 1.2765;
```

Razlika između strukture i unije



```
struct {  
    char sAttr1[5];  
    short sAttr2[3];  
    long sAttr3;  
} struktura_T;
```



```
union {  
    char sAttr1[5];  
    short sAttr2[3];  
    long sAttr3;  
} unija_T;
```

Nizovi

- Uređen skup podataka istog tipa, označen jednim imenom
- Podaci su smesteni kontinualno u memoriji
- Ime = adresa prvog elementa
- Pristup svakom od članova polja se ostvaruje pomoću jednog ili više indeksa, zavisno od dimenzije niza

```
int k[5] = {11, 22, 33, 44, 55};
```

adresa

11	22	33	44	55	
indeks	0	1	2	3	4

k - ime označava početak niza (adresu)
k[i] - vrednost niza na odstojanju i

```
int a[3][4] =  
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};  
...  
{ {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
```

adresa

[0] [0]	[0] [1]	[0] [2]	[0] [3]
[1] [0]	[1] [1]	[1] [2]	[1] [3]
[2] [0]	[2] [1]	[2] [2]	[2] [3]

Kolone

Pokazivači

- Pokazivač je promenljiva čija je vrednost adresa podatka smeštenog u memoriji, izražena u byte-ima
- Pokazivači mogu adresirati
 - Promenljive
 - Funkcije
- Deklaracija pokazivača
 - `tip_pod *pnt;`
- `tip_pod` deklariše format i veličinu podatka na toj adresi
- Uvećanje `(++)` i smanjenje `(--)` pokazivača menja njegovu vrednost za `sizeof(tip_pod)`
- Isto važi i za sabiranje i oduzimanje pokazivača sa celobrojnom vrednošću
 - $[pnt \pm n] = [pnt] \pm n \times sizeof(tip_pod)$

Pokazivači na promenljive

- Postavljanje pokazivača
 - `char *bp = &chr;`
 - `int *ip = NULL; (0L)`
- `*` - sadržaj promenljive tip_pod na adresi iz pokazivača
 - `char t = *bp;`
- void * deklariše neoznačen pokazivač
 - samo adresa – odgovara `unsigned char *`
 - bez warning-a se može puniti pokazivač proizvoljnog tipa
- Pokazivači se mogu eksplicitno cast-ovati kao i promenljive
 - `unsigned j = *(unsigned*) pnt`
- Pokazivač na pokazivač
 - `int **pnt; char **bp -> char *bp[];`
- Niz pokazivača
 - `int *pnt [20];`

Nizovi i pokazivači

- element `Niz[10];`
- element `*pokEl = Niz;`
- $\text{Niz}[3] == *(\text{Niz} + 3) == \text{pokEl}[3] == *(\text{pokEl} + 3)$
 - pristup trećem (četvrtom) elementu niza `Niz`
- $\&\text{Niz}[5] == \text{Niz} + 5 == \&\text{pokEl}[5] == \text{pokEl} + 5$
 - adresa petog (šestog) elementa niza `Niz`
- $(\text{char}^*)\text{Niz} + 7 == (\text{char}^*)\text{pokEl} + 7$
 - adresa sedmog (osmog) bajta zauzetog memorijskog prostora
- $*((\text{char}^*)\text{Niz} + 9) == *((\text{char}^*)\text{pokEl} + 9)$
 - pristup devetom (desetom) bajta zauzetog memorijskog prostora

Pokazivači na funkcije

```
#include <stdio.h>

/* functions' prototypes */
int fun1(int, double);
int fun2(int, double);
int fun3(int, double);

/* an array of a function pointers */
int (*p[3])(int, double);

int main()
{
    int i;

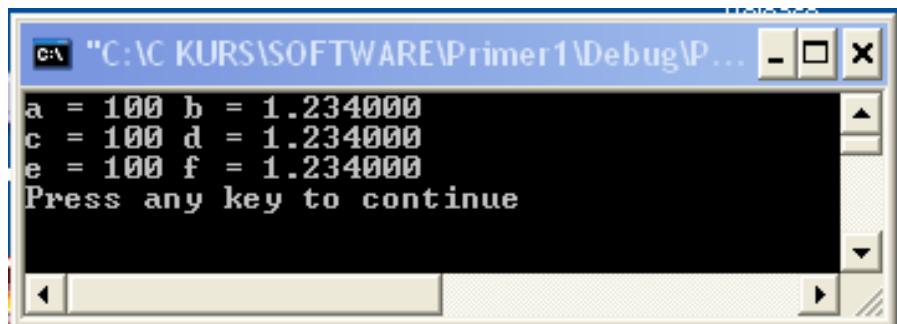
    /* assigning address of functions */
    p[0] = fun1;
    p[1] = fun2;
    p[2] = fun3;

    /* calling with arguments */
    for(i = 0; i <= 2; i++)
    {
        (*p[i])(100, 1.234);
    }
    return 0;
}
```

```
/* functions' definition */
int fun1(int a, double b)
{
    printf("a = %d b = %f", a, b);
    return 0;
}

int fun2(int c, double d)
{
    printf("\nc = %d d = %f", c, d);
    return 0;
}

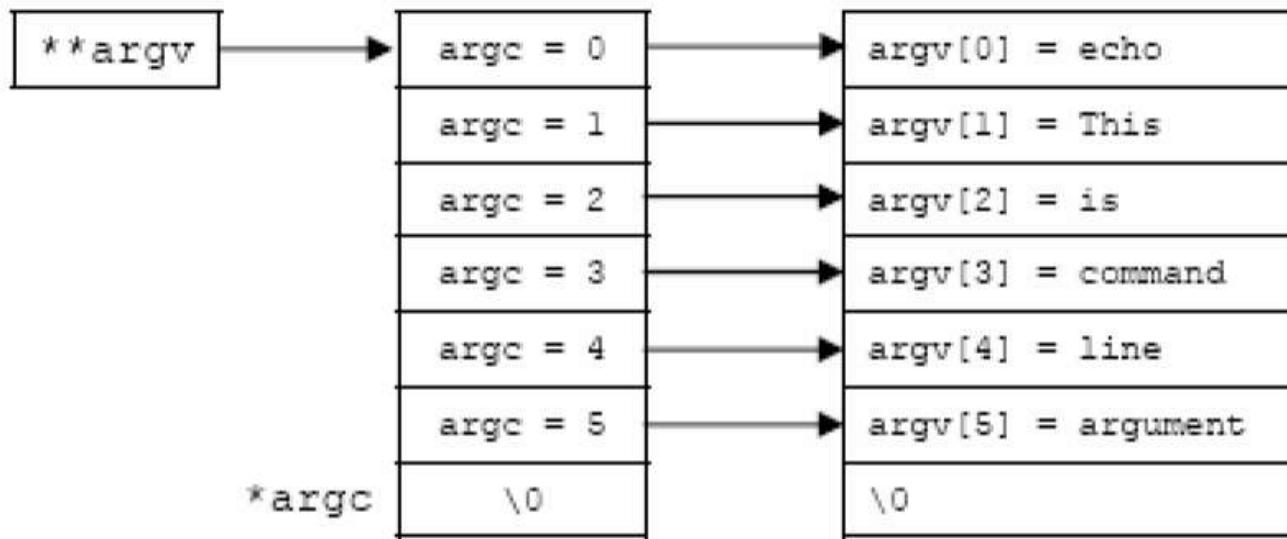
int fun3(int e, double f)
{
    printf("\ne = %d f = %f\n", e, f);
    return 0;
}
```



Main funkcija

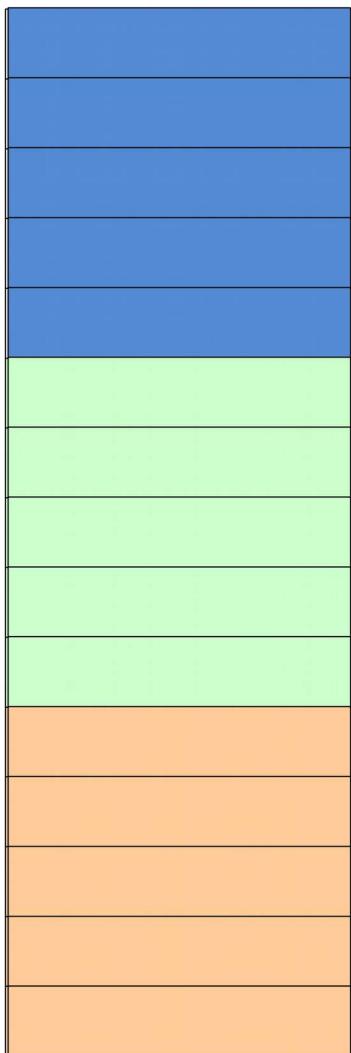
- Ulazna tačka u programu
- int main(int argc, char **argv)

```
C:\>echo This is command line argument  
This is command line argument
```



Višedimenzionalni nizovi

Niz



← pokNiz

```
char Niz[3][5];  
char **pokNiz = (char**)Niz;
```

pokNiz[0] - pokazivač na char

Regularna sekvenca:

```
enum{DIM1=3, DIM2=5};  
char Niz[DIM1][DIM2];
```

char *pokNiz = (char*)Niz;

.....

Niz[2][3] == *(pokNiz + 2 * DIM2 + 3)

Niz

Višedimenzionalni nizovi

long Niz[2][3][6]

long Niz[2][3][6];

long ***pokNiz = (long***)Niz;



OVO NE VALJA

pokNiz[0] - pokazivač na pokazivač na long

Niz[1][2][3] != pokNiz[1][2][3];

Big i Little Endian

- Redosled slaganja byte-ova u memoriji
- BE: najviši zadnji
- LE: najniži zadnji
- Bi-Endian: obe mogućnosti
- Zavisno od HW platforme
 - Intel je LE
 - MIPS je BE (ili LE – opcionalno)
- Problem je prisutan i kod
 - Čitanja binarnih datoteka
 - Komunikacione razmene podataka
- Ne postoji sistemsko rešenje, tj mora se programirati konverzija podataka
- Ograničava prenosivost koda

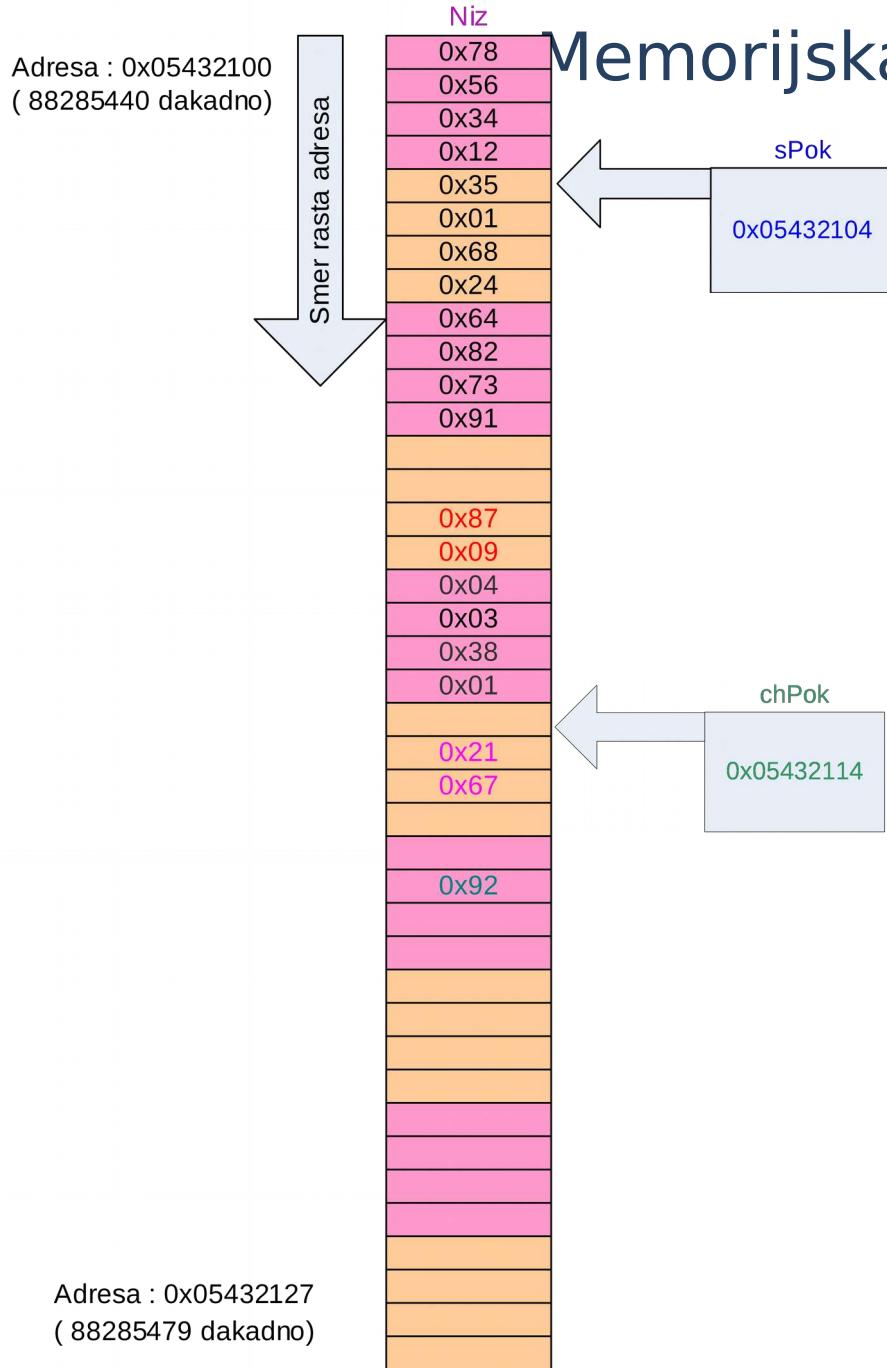
```
char buf[4];
int k = 0x01020304;
memcpy( buf, &k, sizeof(k) );
```

	Little Endian	Big Endian
buf[0]	04	01
buf[1]	03	02
buf[2]	02	03
buf[3]	01	04

Intel MIPS



Memorijska slika - Little Endian



```
long Niz[10];
```

Niz[0] = 0x12345678;

Niz[1] = 0x24680135;

Niz[2] = 0x91738264;

```
short *sPok = (short*)&Niz[1]
```

sPok[5] = 0x0987

```
char *chPok = (char*)(sPok + 8);
```

chPok[1] = 0x21;

chPok[2] = 0x67;

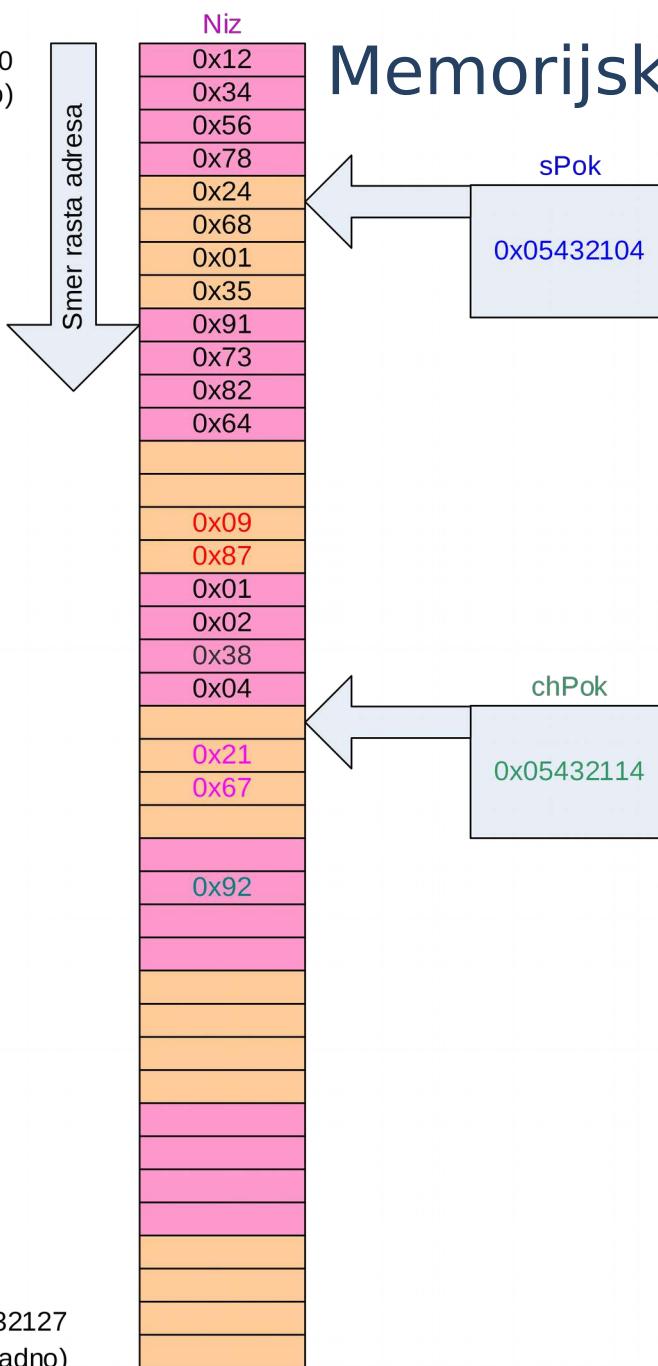
`*(chPok+5) = 0x92;`

***(Ni_z + 4) = 0x01020304;**

```
*((char*)Niz + 18) = 0x38;
```

Niz[4] == 0x01380304

Adresa : 0x05432100
(88285440 dakadno)



Memorijska slika – Big Endian

long Niz[10];

Niz[0] = 0x12345678;

Niz[1] = 0x24680135;

Niz[2] = 0x91738264;

short *sPok = (short*)&Niz[1];

sPok[5] = 0x0987;

char *chPok = (char*)(sPok + 8);

chPok[1] = 0x21;

chPok[2] = 0x67;

*(chPok+5) = 0x92;

*(Niz + 4) = 0x01020304;

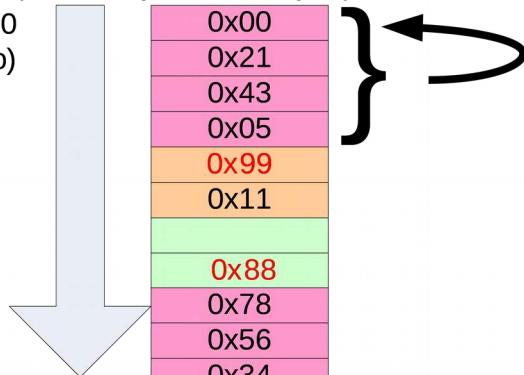
((char)Niz + 18) = 0x38;

Niz[4] == 0x01023804

Poravnavanje strukture

A sa poravnajem na 8 bajta (default u MVC)

Adresa : 0x05432100
(88285440 dakadno)



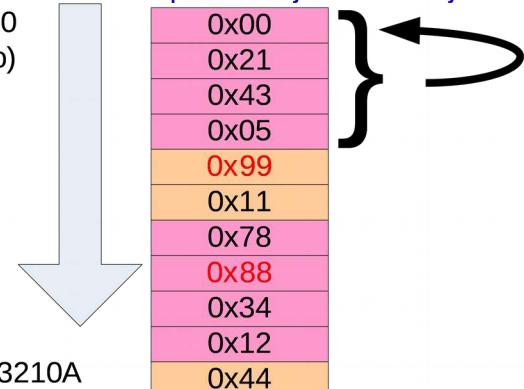
Adresa : 0x0543210F
(88285455 dakadno)

```
typedef struct
{
    unsigned char * p1;
    unsigned short p 2;
    unsigned long  p 3;
    unsigned char  p 4;
} struktura _T;
```

```
struktura _T A;
A.p1 = (unsigned char *)&A.p1;
A.p2 = 0x1122;
A.p3 = 0x12345678;
A.p4 = 0x44;
```

A sa poravnajem na 1 bajt

Adresa : 0x05432100
(88285440 dakadno)

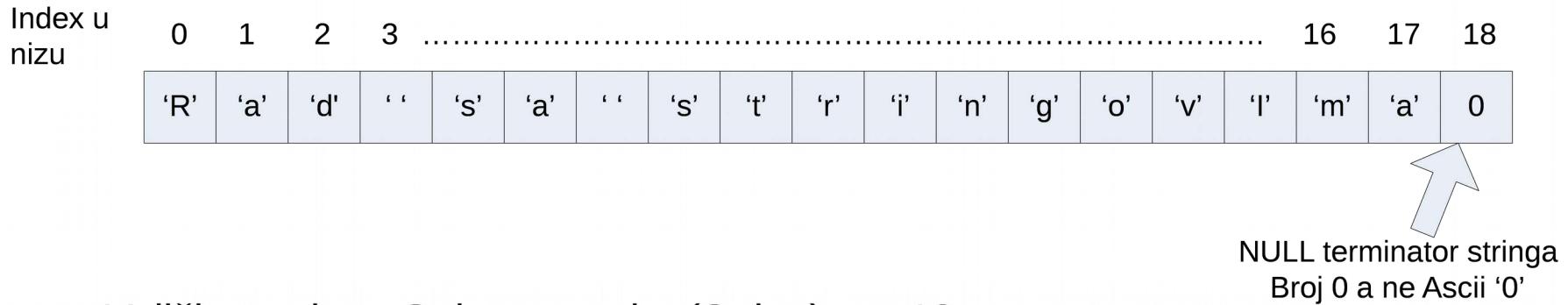


Adresa : 0x0543210A
(88285450 dakadno)

A.p1[4] = 0x99;
*(A.p1 + 7) = 0x88;

Stringovi

char String[] = "Rad sa stringovima";



Veličina srtิงa String == strlen(String) == 18

Veličina zauzetog memorijskog prostora za String == strlen(String) + 1 == 19

char String2[40] = "Rad sa stringovima";

Veličina srtингa String2 == strlen(String2) == strlen(String) == 18

Veličina zauzetog memorijskog prostora za String2 == 40

Stringovi

```
char String[] = "Rad sa stringovima";
```

Index u
nizu

0	1	2	3	16	17	18									
'R'	'a'	'd'	' '	's'	'a'	' '	's'	't'	'r'	'i'	'n'	'g'	'o'	'v'	'l'	'm'	'a'	0



NULL terminator stringa
Broj 0 a ne Ascii '0'

```
char *string3 = String + 4;
```

```
string3 == "sa stringovima"
```

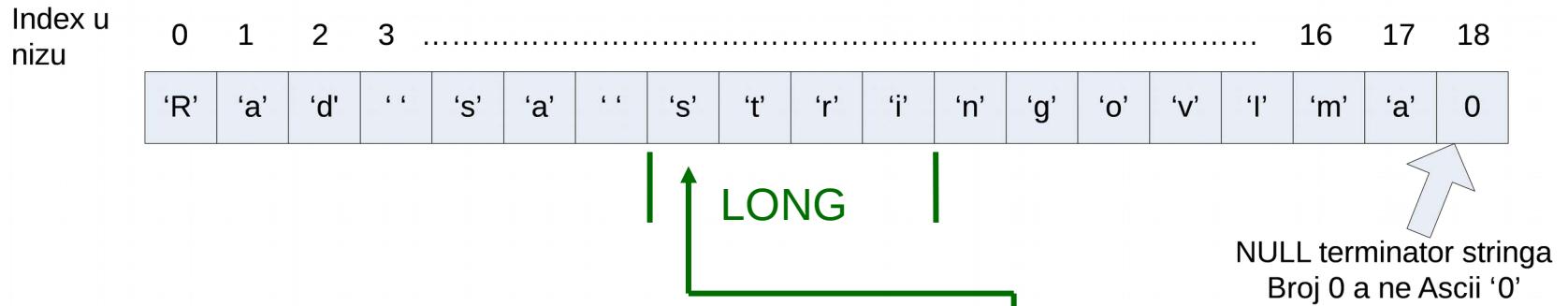
```
string3[9] = 0;
```

NULL terminator stringa

```
string3 == "sa string"
```

```
String == "Rad sa string"
```

Stringovi



Svaki znak stringa je jedan bajt i kako ćemo mi njega videti je samo stvar interpretacije

long *Lstr = (long *)(String + 7);

's' == 0x73
't' == 0x74
'r' == 0x72
'i' == 0x69

*Lstr == 0x69727473

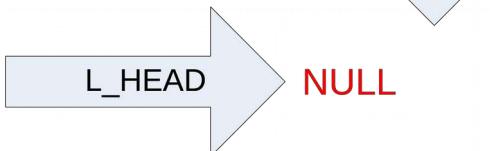
Lstr[2] == 0x00616D69

Jednostruko spregnute Liste

ID == 2
Val == 10
Next == NULL

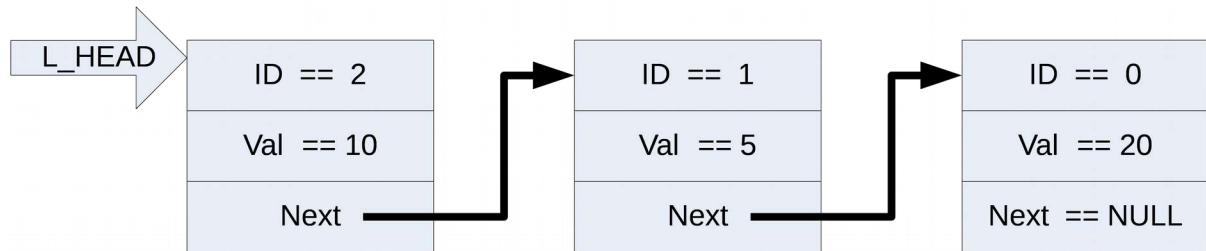
ID == 1
Val == 5
Next == NULL

ID == 0
Val == 20
Next == NULL



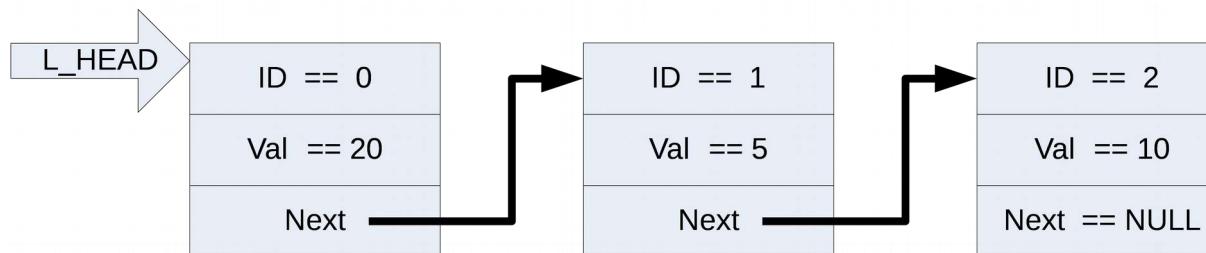
Dodavanje na početak

- ako skidamo elemente samo sa početka onda je ovo jedna realizacija LIFO reda



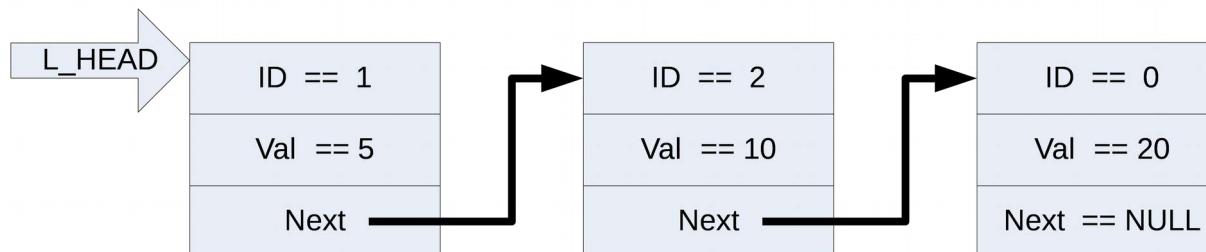
Dodavanje na kraj

- ako skidamo elemente samo sa početka onda je ovo jedna realizacija FIFO reda



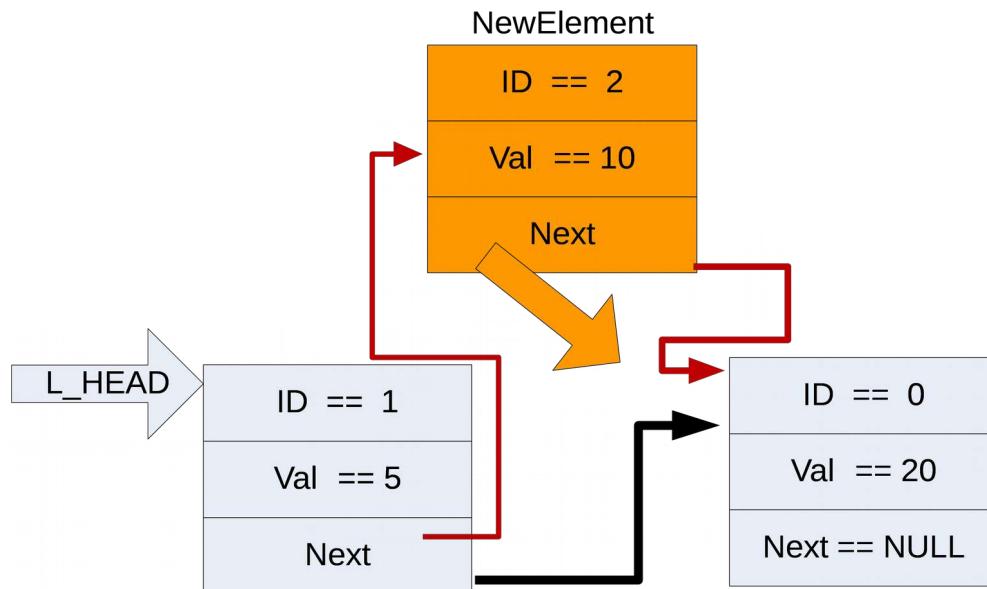
Dodavanje sortirano po Val

- ovo su specifične liste koje se koriste u specifične svrhe
- Val može imati bilo kakvo značenje: prioritet procesa, vreme,...



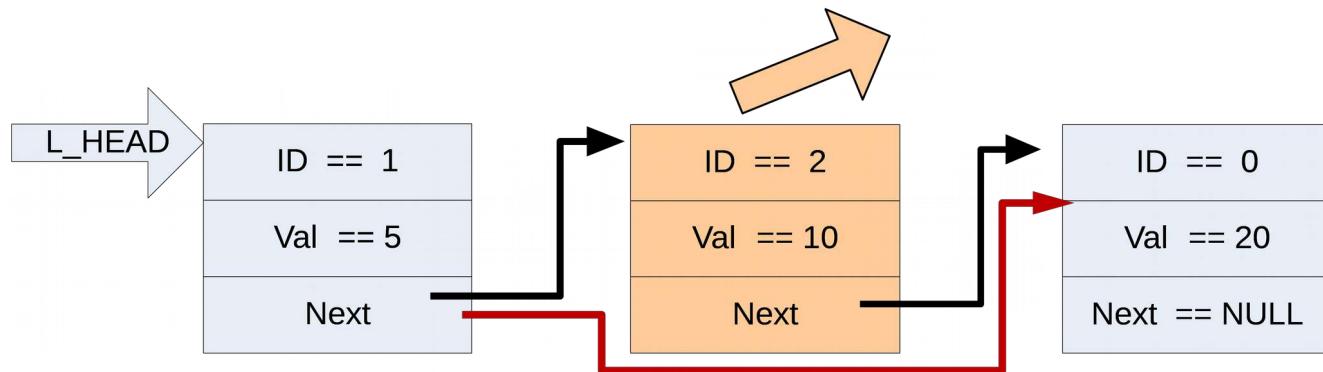
Dodavanje elementa u listu

- Pronaći mesto u listi gde će novi element biti dodat
- Povezati novi element sa sledećim u listi
- Razvezati postojeću vezu prethodnog elementa u listi sa sledećim elementom i povezati ga sa novim elementom



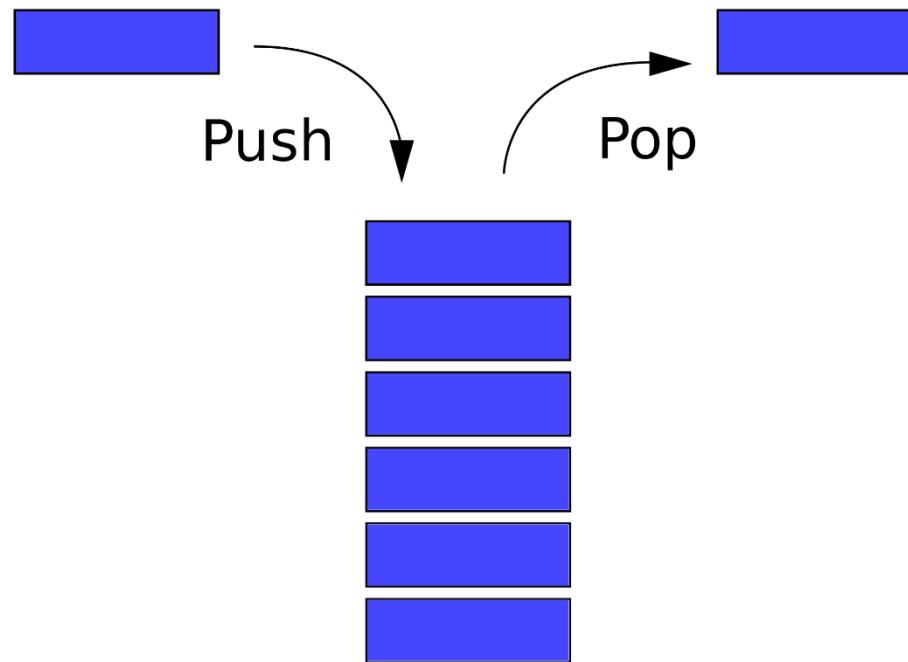
Skidanje određenog elementa iz liste

- Identifikovati (pronaći) element koji treba izbaciti iz liste
- Razvezati postojeću vezu prethodnog elementa u listi sa tekućim (pronađenim) i prethodni element povezati sa sledećim elementom



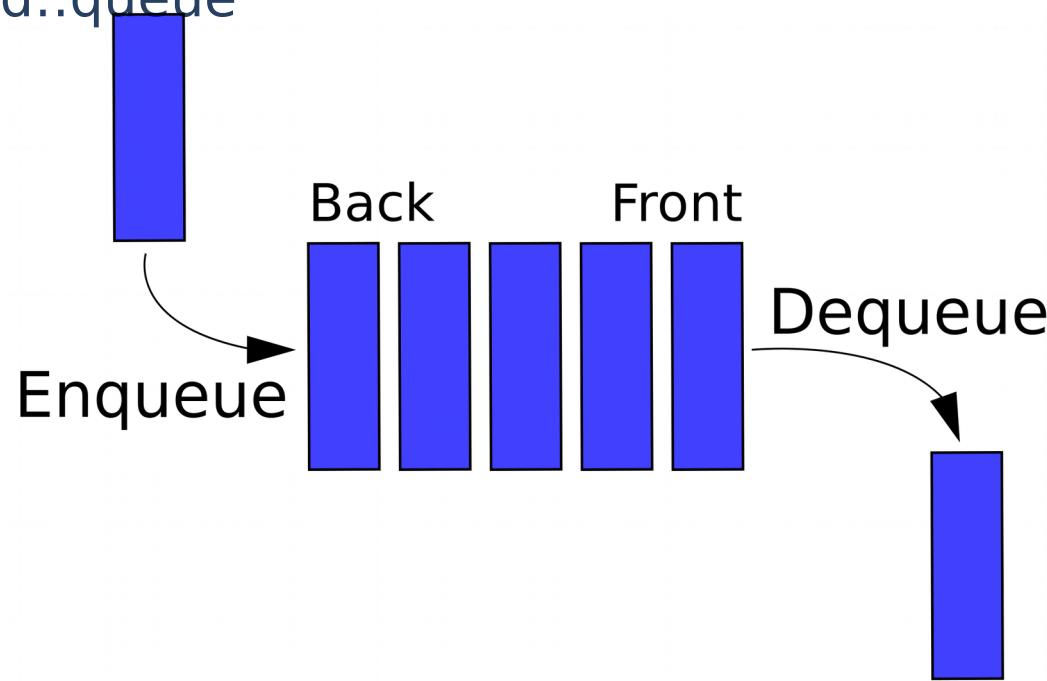
Stack

- Struktura podataka bazirana na principu LIFO (last in, first out)
- Osnovne operacije: push, pop, top
- STL/CLR std::stack



Queue (red)

- Struktura podataka bazirana na principu FIFO - first in, first out
- Osnovne operacije: enqueue, dequeue
- STL/CLR std::queue



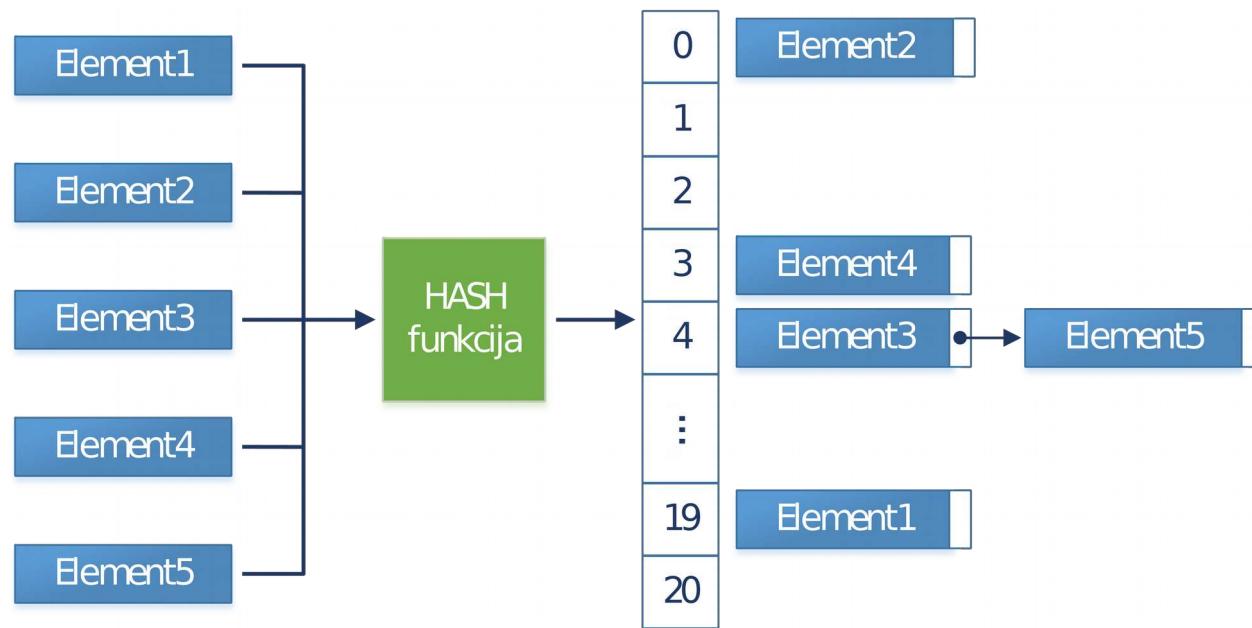
Set (skup)

- Set predstavlja implementaciju matematičkog koncepta konačnog skupa
- Ne sadrži duplike elemenata
- Ne postoji određen redosled elemenata
- Implementacija pomoću različitih struktura podataka:
 - lista
 - stabla
- STL/CLR std::set

HashSet

- Predstavlja implementaciju matematičkog skupa kao i SET
- Hash funkcija određuje indeks elementa
 - $\text{index} = \text{HashFunction(element)} \% \text{HastSetSize}$
 - Efikasno pronalaženje elemenata $O(1)$
- Kolizija se može rešiti ulančavanjem elemenata na istom indeksu
- STL/CLR `std::hash_set`

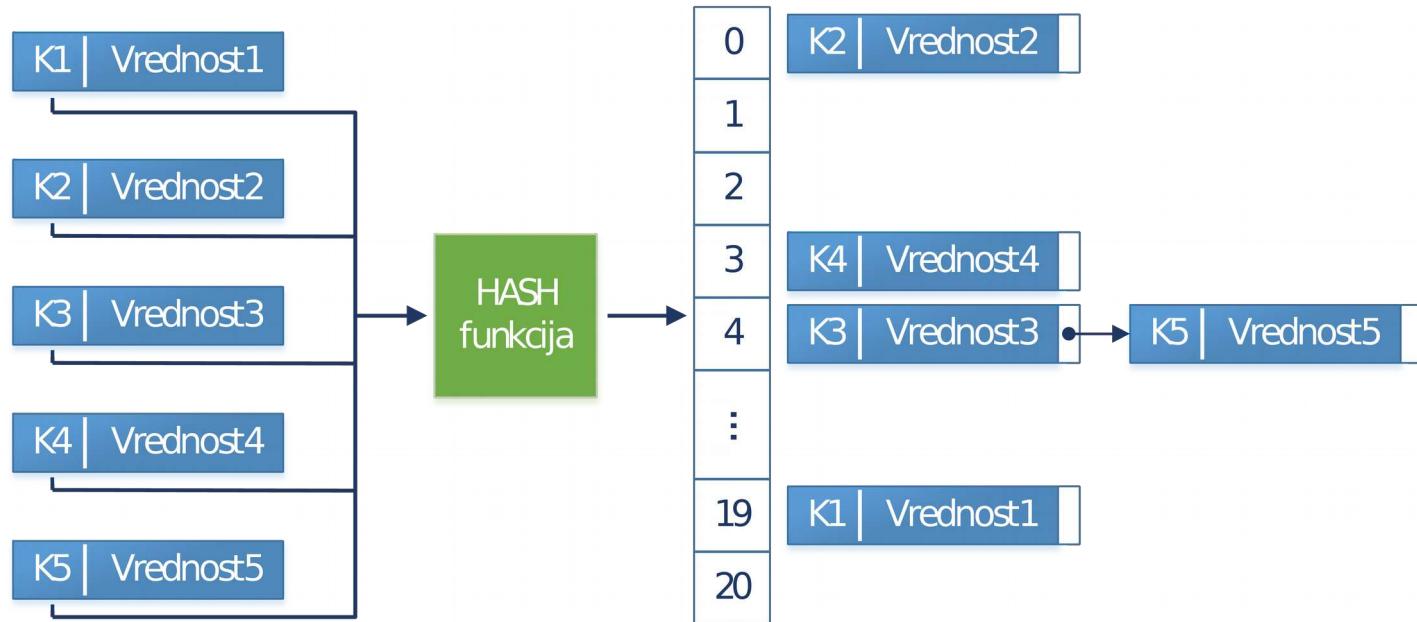
HashSet



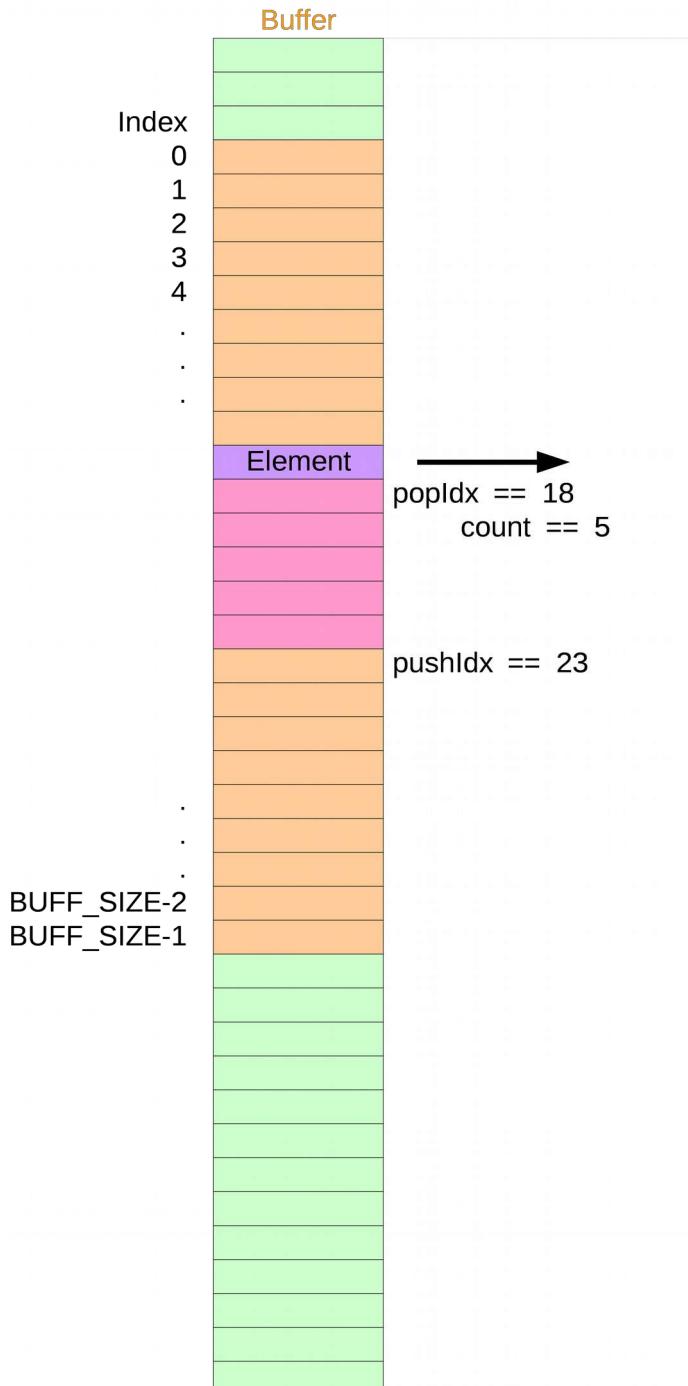
HashMap (HashTable)

- Predstavlja implementaciju asocijativnog niza (mapa, rečnik)
 - skup parova ključ-vrednost
- Svaki element ima jedinstven ključ (struktura ne sadrži duplike ključeva)
- Hash funkcija nad ključem određuje indeks elementa
- Kolizija se može rešiti ulančavanjem elemenata na istom indeksu
- STL/CLR std::hash_map

HashMap (HashTable)



Kružni bafer



```
element Buffer[BUFF_SIZE];
int pushIdx = 0;
int popIdx = 0;
int count = 0;
```

Dodavanje elementa u kružni bafer:

```
if(count == BUFF_SIZE)
    return false;

Buffer[pushIdx] = newEl;
pushIdx++;

if(pushIdx == BUFF_SIZE)
    pushIdx = 0;
count++;
```

Uzimanje elementa iz kružnog bafera:

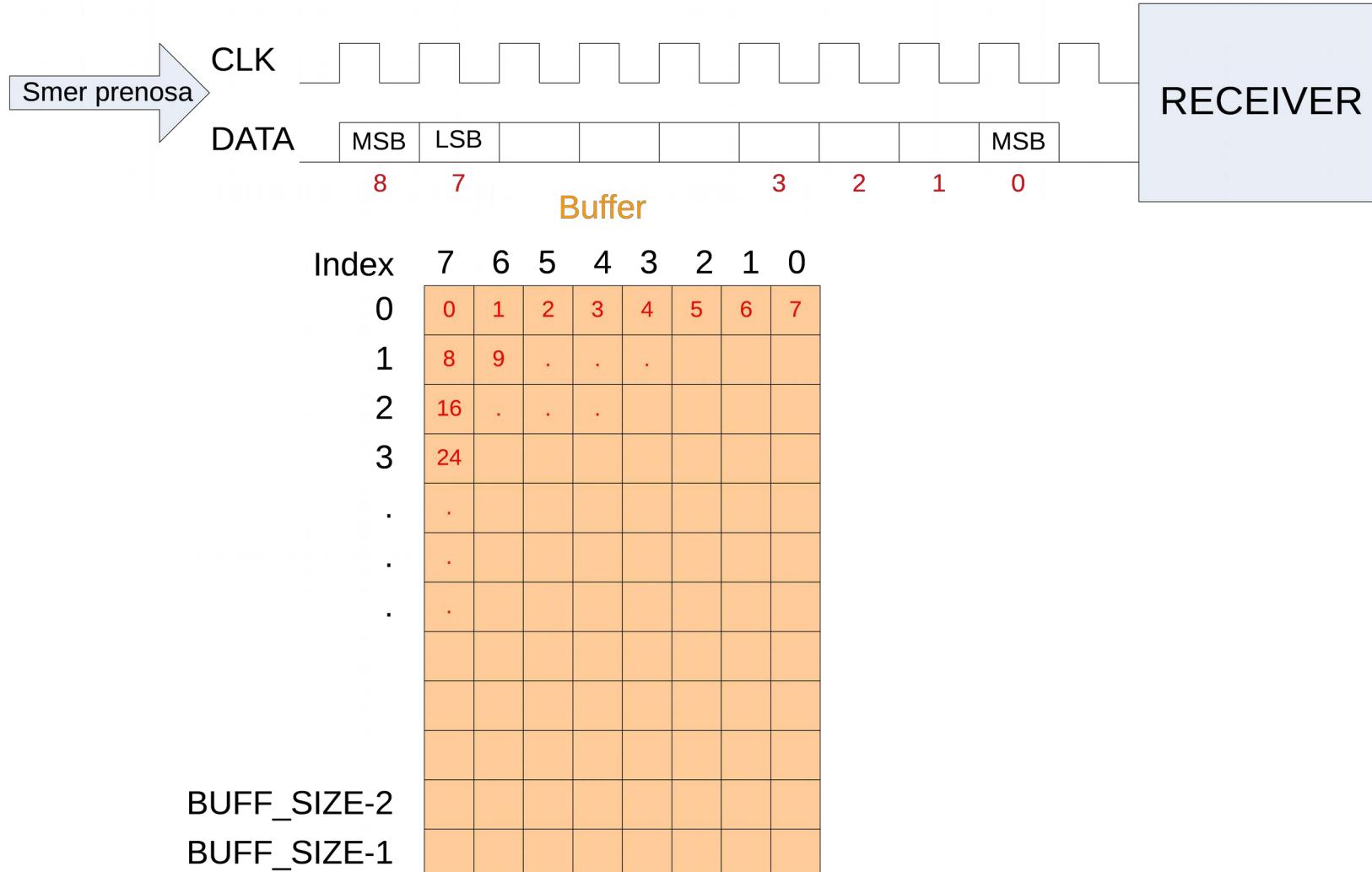
```
if(count == 0)
    return false;

Element = Buffer[popIdx];
popIdx++;

if(popIdx == BUFF_SIZE)
    popIdx = 0;
count--;
```

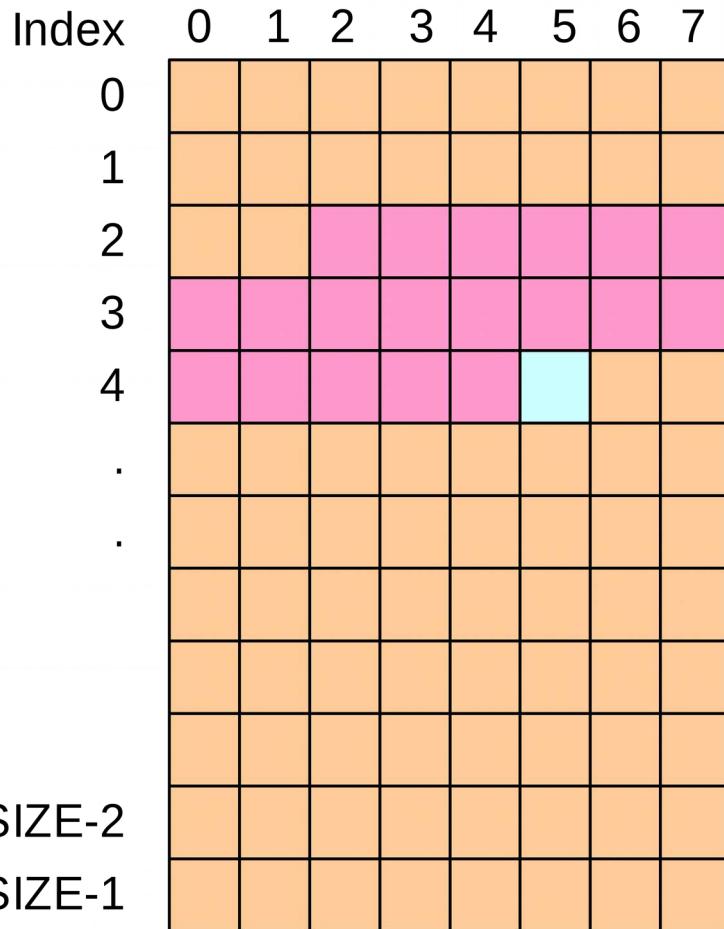
Bitski bafer

Sinhrona serijska komunikacija



Bitski bafer

Buffer



```

popIdx == 18
count == 20
.....
pushIdx == 38
== 0x26
== 0010 0110 b
bajlIdx == 00100 b
== 4
bitIdx == 0x110 b
== 6

```

```

#define BUFF_SIZE 256
#define BUFF_BIT_SIZE 2048
BYTE Buffer[BUFF_SIZE];
int pushIdx = 0;
int popIdx = 0;
int count = 0;

Dodavanje bita u kružni bafer:
if(count == BUFF_BIT_SIZE)
    return false;
int bajlIdx = pushIdx >> 3;
int bitIdx = pushIdx & 0x00000007;
BYTE bitMask = 0x80 >> bitIdx;
if(Bit == 1)
    Buffer[bajlIdx] |= bitMask;
else
    Buffer[bajlIdx] &= ( ~bitMask );

pushIdx++;
if(pushIdx == BUFF_BIT_SIZE)
    pushIdx = 0;
count++;

```

Buffer (memory) pool

- Jednom se zauzme određen broj bafera fiksne dužine, pa se kasnije koriste bez ponovnog alociranja memorije
- Bafer predstavlja niz bajtova
- Lista sadrži adresu svakog bafera
- Iz liste se uklanja adresa kada se bafer koristi
- U listu se doda adresa na bafer koji se više ne koristi

