



Знаковни низови - стрингови



Знаковни низови у Цеу



- Знаковни низови нису посебан тип података.
- **Знаковни низ је низ char типа који се завршава са '\0'.**
- Неколико синтаксних специфичности и подршка у стандардним библиотекама чине знаковни низ посебним елементом Це језика
- Све остало је на програмеру.
- Зато морамо бити пажљиви, јер рад са знаковним низовима је извор многих проблема.

Пример:

```
char buffer[21];
```

Заузима меморију за знаковни низ у који стаје 20 знакова.



Синтаксне посебности

- Постојање стринг литерала/константи.

`"ovo je string literal"`

- Спајање (или надовезивање, никако ~~конкатенација~~) стринг литерала

`"string" " literal" " sa" " odvojenim" " recima"`

`"string literal sa odvojenim recima"`

Корисно, рецимо, када се прелази у нови ред.

- Иницијализација

```
char string[] = {1, 2, 3, 4, 5};
```

```
char string[] = {'a', 'b', 'c', 'd', 'e', '\0'};
```

```
char string[] = "abcde";
```

```
char* string = {1, 2, 3, 4, 5};
```

```
char* string = {'a', 'b', 'c', 'd', 'e', '\0'};
```

```
char* string = "abcde";
```



Знаковни литерал наспрам стринг литерала



Знаковни литерал је под једноструким наводницима (' , не ").

```
char buf[10];  
buf[0] = 'A'; /* correct */  
buf[0] = "A"; /* incorrect */  
buf[1] = '\0'; /* NULL terminator */
```

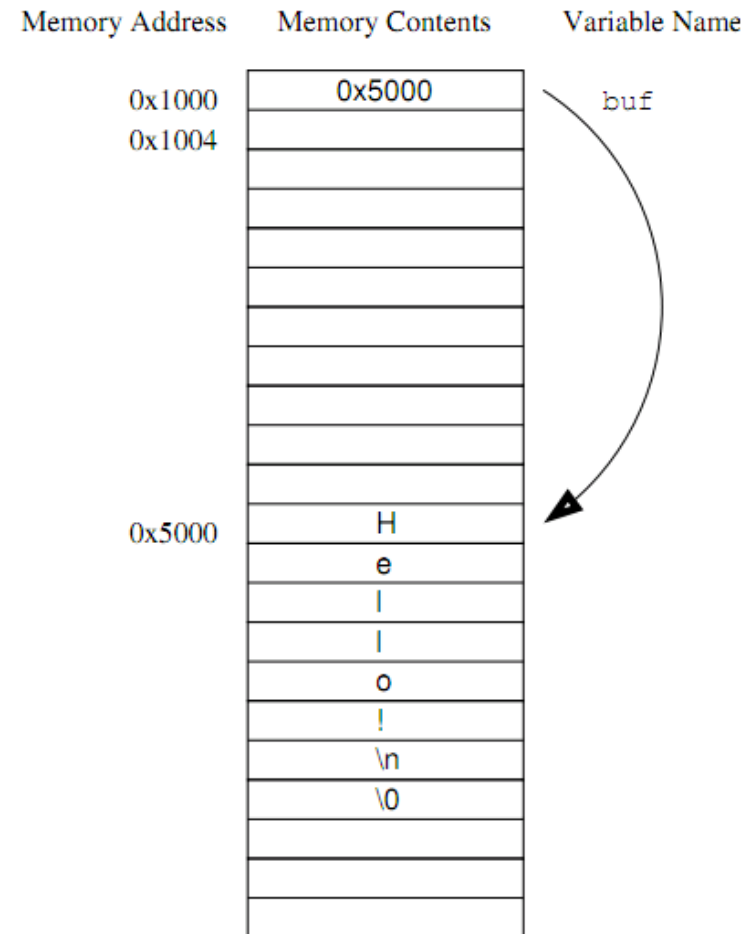


Пример



```
char* buf = "Hello!\n";
```

- Променљива **buf** је показивач на меморију где се стринг налази.
- Приметити NULL ('\0') знак на крају - аутоматски је додат.





Библиотечке посебности

- \0 на крају је битно само зато што то библиотечке функције очекују.
- Посебни симбол у формат стрингу за printf и scanf (и сродне функције).

```
char str[] = "Nesto";  
int i;  
printf("%s", str); // sta ako nema \0 na kraju?  
scanf("%d%s", &i, str); // sta ako vise od 5 znakova?
```

- Библиотечке функције за рад са знаковним низовима
 - Пре свега string.h
 - stdlib.h
 - stdio.h



Копирање стрингова

```
char* buf1 = "Hello";
char* buf2 = "olleH";
buf2 = buf1;
buf2[2] = 'M';
printf("%s %s", buf1, buf2);
Runtime error!
```

```
char* buf1 = "Hello";
char buf2[100];
buf2 = buf1;
```

Compile error!

```
#include <string.h>
char* buf1 = "Hello";
char buf2[100];
strcpy(buf2, buf1);
buf2[2] = 'M';
printf("%s %s", buf1, buf2);
Output: Hello HeMlo
```



Пример



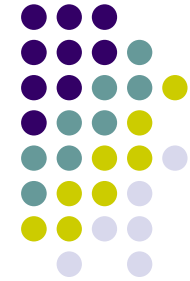
```
char buf[] = "Hello, World!\n";
char* buf2 = buf + 7;
printf("buf: %s\n", buf);
printf("buf2: %s\n", buf2);
buf2[0] = 'M';
printf("buf: %s\n", buf);
```

Шта је излаз?

Memory Address	Memory Contents	Variable Name
1000	5000	buf buf2
1004	5007	
5000	H	buf buf2
5001	e	
5002	l	
5003	l	
5004	o	
5005	,	
5006	,	
5007	W	
5008	o	
5009	r	
5010	l	
5011	d	
5012	!	
5013	\n	
5014	\0	
5015		
5016		
5017		
5018		



Премашивање бафера



Стринг не расте сам по потреби. Бафер (парче меморије) који му је додељен се не мења.

Пример:

```
char s1[] = "1. string";  
char s2[] = "2. string";  
strcpy(s1, "This string is too long!\n");
```

Копирамо стринг величине 25 у меморију која је предвиђена за 9 знакова!

Врло је могуће да смо преписали s2!

Шта више, пошто смо премашили и s2 почели смо са писањем по ко зна чему.

Компајлер неће ово приметити, а и често неће бити детектовано ни током извршавања (осим што програм неће радити)!

Memory Address	Memory Contents	Variable Name
1000	5000	s1
1004	5010	
		s2
5000	T	s1
5001	h	
5002	i	
5003	s	
5004	' '	
5005	s	
5006	t	
5007	r	
5008	i	
5009	n	
5010	g	
5011	' '	
5012	i	
5013	s	
5014	' '	
5015	t	
5016	o	
5017	' '	
5018	l	



Стринг литерали

Пример 1

```
char* str;  
str = "hello";  
printf("%s\n", str);
```

Пример 2

```
char str[100];  
strcpy(str, "hello");  
printf("%s\n", str);
```

Исти испис на екран, али понашање врло различито.



Стринг литерали

Пример 1

```
char* str;  
str = "hello";  
printf("%s\n", str);  
strcpy(str, "hello");
```

Пример 2

```
char str[100];  
strcpy(str, "hello");  
printf("%s\n", str);  
str = "hello";
```

Пример 1 узрокује упис на недозвољено место.
Пример 2 се неће ни превести.



Још један поучан пример



```
int main()
{
    char* str;
    str = (char*)malloc(100);
    str = "hello";
    free(str);
    return 0;
}
```

```
int main()
{
    char* str;
    str = (char*)malloc(100);
    strcpy(str, "hello");
    free(str);
    return 0;
}
```

Леви пример се преводи исправно али приликом извршавања пријављује грешку. Зашто?



Стрингови као параметри функције



Као и редовни низови, стрингови се преносе само „преко референце”.

```
void Print1(char* str)
{
    printf("%s", str);
}
```

```
void Print2(char* ary, int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%c", ary[i]);
}
```



<string.h> неке важније функције



```
char* strcpy(char* s1, const char* s2);
```

```
char* strncpy(char* s1, const char* s2, size_t n);
```

```
char* strcat(char* s1, const char* s2);
```

```
char* strncat(char* s1, const char* s2, size_t n);
```

```
int strcmp(const char* s1, const char* s2);
```

```
int strncmp(const char* s1, const char* s2, size_t n);
```

```
char* strtok(char* str, const char* delim);
```

Погледати у тексту стандарда детаљни опис ових функција.



<string.h> још функција



```
void* memcpy(void* s1, const void* s2, size_t n);  
void* memmove(void* s1, const void* s2, size_t n);  
  
int memcmp(const void* s1, const void* s2, size_t n);  
  
void* memset(void* str, int c, size_t n);
```



Функције за конверзију



Из знаковоног низа у бројеве. <stdlib.h>

```
int atoi(const char* nptr);  
long atol(const char* nptr);  
long long atoll(const char* nptr);  
double atof(const char* nptr);
```

Обрнуто? <stdio.h>

```
int sprintf(char* s, const char* format, ...);  
  
sprintf(s, "%d", 5); // s = "5"
```




Поравнање података



Поравнање

- За меморијску адресу се каже да је поравната на n бајтова уколико је она умножак броја n .
- Многе физичке архитектуре намећу захтеве за поравнањем адреса одређених објеката зарад потпуне искоришћености могућности које нуде.
- Дакле, приступ и непоравнатим подацима је увек могућ, али није ефикасан.
- Пример: ширина меморије је 8 бита, а меморији је омогућен 32битни приступ - али само са адреса које су умножак четворке.



Уобичајена поравнања

- Уобичајена поравнања за 32битну архитектуру x86:
 - **char** (један бајт) поравнат на **1 бајт**.
 - **short** (два бајта) поравнат на **2 бајта**.
 - **int** (четири бајта) поравнат на **4 бајта**.
 - **float** (четири бајта) поравнат на **4 бајта**.
 - **double** (осам бајтова) поравнат на **8 бајтова** под оперативним системом Windows, поравнат на **4 бајта** на о.с. Linux (посебном компајлерском опцијом се поравнање подесити на 8 бајтова).
 - **показивач** (четири бајта) поравнат на **4 бајта**

```
char a;  
int b;  
char c;  
short d;
```



Шта ако подаци нису правилно поравнати?



- Компајлер претпоставља правилно поравнање и сам ће се за њега постарати, али у одређеним случајевима програмерском непажњом може доћи до непоравнатог приступа меморији.
- У случају непоравнатог приступа може се десити следеће:
 - Програм неће дати добар резултат
 - Ако приступ меморији иде кроз оперативни систем, онда ће се можда он постарати да се све, иако неефикасније, ипак добро заврши

```
int8_t buffer[4];  
int32_t x = *(int32_t*)buffer;
```



Поравнање чланова структуре



```
char a;          struct S          x = sizeof(struct S);
int b;           {
char c;           char a;          y =
short d;          int b;          sizeof(a) + sizeof(b) +
                  char c;          sizeof(c) + sizeof(d);
                  short d;
                  };          y ≤ x
```

- Код структуре чланови морају бити у меморији поређани редом којим су наведени.
- Величина променљиве типа struct S је већа од суме појединачних величина њених чланова.
- То је зато што долази до уметања бајтова између чланова да би се обезбедило њихово одговарајуће поравнање.



Још један пример уметања



```
struct MixedData    struct MixedData
{
    char Data1;      {
                        char Data1;
                        char Pad1[1];
                        short Data2;
                        short Data2;
                        int Data3;
                        int Data3;
                        char Data4;
                        char Data4;
                        char Pad2[3];
    };
};
```

- Иако би само са једним уметањем (Pad1) сви елементи понаособ били поравнати, ипак се бајтови умећу и на крају (Pad2) да би се обезбедило правилно поравнање и у случају низа структура.
- Дакле, величина структуре ће увек бити умножак поравнања највећег основног типа њених чланова.



Једнакост структура

```
struct MixedData s1 = {a, b, c, d};
```

```
struct MixedData s2 = {a, b, c, d};
```

```
/*s1 = s2;*/
```

```
/*s1 = s2;*/
```



```
if (s1 == s2)
```



```
if (memcmp(&s1, &s2, sizeof(struct MixedData)) == 0)
```

```
memcpy(&s1, &s2, sizeof(struct MixedData));
```



```
if (memcmp(&s1, &s2, sizeof(struct MixedData)) == 0)
```

```
memcpy(&s1, &s2, sizeof(struct MixedData));
```



```
if (s1 == s2)
```



Паковање структура

- Баратање са поравнатим структурама и њиховим елементима је брже, али уметнути бајтови повећавају утрошак меморије.
- Смањење утрошка меморије без успорења рада може се остварити променом редоследа елемената у структури.
- Стандард гарантује редослед елемената, тако да компајлер не може самоиницијативно преуредити структуру, али програмер може.
- Међутим, такво преуређење може само донекле смањити величину. Ако желимо још мање меморије да утрошимо, то можемо урадити, али по цену брзине.
- Већини Це компајлера је могуће наредити да упакују елементе структуре на жељено поравнање, нпр. `pack(2)` значи да компајлер треба да поравна податке на максимално 2 бајта, што значи да уметнута поља неће бити већа од једног бајта.
- Паковање структура се најчешће користи за смањење утрошка меморије, али може се користи и за припрему података за мрежно слање и слично.



Преуређење елемената структуре



Пример структуре `MixedData`. Лево са преуређеним елементима, а десно је дата првобитна структура:

<pre>struct MixedData { char Data1; char Data4; short Data2; int Data3; };</pre>	<pre>struct MixedData { char Data1; short Data2; int Data3; char Data4; };</pre>
--	--

Са оваквим преуређењем више није потребно уметнути ни један бајт.

Подсећање: `char` - 1 бајт, `short` - 2 бајта, `int` - 4 бајта на x86

Колико бајтова заузимају лева и десна структура?



Паковање структуре

- Замислимо да структура `MixedData` нема поље `Data 2`. У том случају ни приликом најбољег редоследа чланова структуре не би могли имати мање од два уметнута бајта.
- Тада се можемо ослонити на могућност коју многи компајлери нуде, али није део стандарда, а то је паковање структуре. Већина компајлера (Microsoft, Borland, GNU...) користе `#pragma` директиве као механизам задавања жељеног паковања.

```
#pragma pack(push)    /* push current alignment to stack */
#pragma pack(1)        /* set alignment to 1 byte boundary */
struct MixedData
{
    char Data1;
    int Data3;
    char Data4;
};
#pragma pack(pop)      /* restore original alignment from stack */
```



GNU `__attribute__` проширење



- У GCC-у је уведена нова кључна реч `__attribute__` која омогућава да се одређене особине придруже променљивама (укључујући и поља структуре) и функцијама.
- Дат је код који превођен GCC-ом даје исти резултат као и код на претходном слајду:

```
struct MixedData
{
    char Data1;
    int Data3;
    char Data4;
}__attribute__ ((packed));
```

- У случају потребе за посебним поравнањем, које не произилази из потреба за поравнањем основних типова, у GCC-у је могуће искористити `aligned` атрибут. Њиме се наређује компајлеру да одређену променљиву поравна на задат начин.

```
int data __attribute__ ((aligned (16))) = 0;
```

Компајлер ће променљиву `data` поставити на адресу која је умножак броја 16.